《程序设计与算法》——《C++ 程序设计》学习笔记

Du Ang

du2ang233@gmail.com

2017 年 9 月 4 日

目录

1 C 语言中的高级内容——从 C 走进 C++

1.1 函数指针

1.1.1 函数指针的定义

程序运行期间，每个函数都会占用一段连续的内存空间。而函数名就是该函数所占内存区域的起始地址（也称“入口地址”）。将入口地址赋给一个指针变量，那该指针就指向了这个函数，这种指向函数的指针变量称为“函数指针”。我们可以通过函数指针来调用函数。

定义函数指针：类型名 (* 指针变量名)(参数类型 1, 参数类型 2, ...)

例如：`int (*pf)(int, char);` 表示 `pf` 是一个函数指针，它所指向的函数返回类型是 `int`，该函数应有两个参数，第一个是 `int` 类型，第二个是 `char` 类型。

1.1.2 函数指针和 qsort 库函数

为什么要定义函数指针呢？干嘛不直接用函数名来调用？

如果要对数组排序，需要知道：

- 数组的起始地址
- 数组元素的个数
- 每个元素的大小（由此可以算出每个元素的地址）
- 元素谁在前谁在后的规则

C 语言快速排序库函数可以对任意类型的数组进行排序：

```
void qsort(void *base, int nelem, usigned int width,
    int (*pfCompare)(const viod *, cont void *));
```

- `base`: 待排数组的起始地址
- `nelem`: 带排序数组的元素个数
- `width`: 待排序数组的每个元素的大小（以字节为单位）
- `pfCompare`: 比较函数（程序员自己编写）的地址

```
int compare(const void *elem1, const void *elem2);     // 比较函数
```

比较函数的编写规则：

1. 如果 `*elem1` 应该排在 `*elem2` 前面，则函数返回值是负整数
2. 如果 `*elem1` 和 `*elem2` 哪个排在前面都行，那么函数返回 0
3. 如果 `*elem1` 应该排在 `*elem2` 后面，则函数返回值是正整数

一个比较函数的示例：

```
int myCompare(const void *elem1, const void *elem2)
{
    unsigned int *p1, *p2;
    p1 = (unsigned int *) elem1;    // 只写“*elem1” 非法，编译器不知道其类型和大小
    p2 = (unsigned int *) elem2;
    return (*p1 % 10) - (*p2 % 10); // 个位数小的排在前面
}
```

1.2 命令行参数

将用户在命令行窗口输入可执行文件名的方式启动程序时，跟在可执行文件后面的那些字符串，称为“命令行参数”。命令行参数可以有多个，以空格分隔。

```
int main(int argc, char *argv[])
{
    ...
}
```

argc: 代表程序启动时，命令行参数的个数。执行程序本身的文件名是第一个命令行的参数。argv: 指针数组，其中的每个元素都是一个 char * 类型的指针，该指针指向存放命令行参数的字符串。

```
int main(int argc, char const *argv[]) {
    for (int i = 0; i < argc; i++) {
        cout << argv[i] << " ";
    }
    cout << endl;
    return 0;
}
```

如果参数中包含空格，需要将参数用 "..." 引起来
输入：./CmdArguments hello world "hello world"
输出：./CmdArguments hello world hello world

1.3 位运算

位运算：对于整数类型（int、char、long 等）变量中的某一位（bit），或者若干位进行操作。
C/C++ 语言提供了六种位运算符来进行位运算操作：

- & 按位与（双目）
- | 按位或（双目）
- ^ 按位异或（双目）
- ~ 按位非（取反）（单目）
- << 左移（双目）
- >> 右移（双目）

1.3.1 按位与 &

通常用按位与来将某变量中的某些位清 0 且同时保留其他位不变。

```
n = n & 0xffffff00; // 将 int 类型的 n 的低 8 位全部置 0
n &= 0xffffff00;
n &= 0xff00;        // 将 short 类型的 n 的低 8 位全部置 0
```

也可以用按位与来获取某变量中的某一位。例如我们想判断 int 型变量 n 的右数第 7 位是否为 1，只需看表达式 n & 0x80 的值是否等于 0x80 即可。

1.3.2 按位或 |

按位或通常用来将某变量中的某些位置 1 且保持其他位不变。

```
n |= 0xff;  // 将 int 型变量 n 的低 8 位全置成 1
```

1.3.3 按位异或 ^

按位异或运算通常用来将某变量中的某些位取反，且保持其他位不变。

```
n ^= 0xff;  // 将 int 型变量 n 的低 8 位取反
```

异或运算的特点：如果 $a \hat{} b = c$，那么 $c \hat{} b = a$，$c \hat{} a = b$。（穷举法可证）
可以利用此规律用来进行最简单的加密和解密。
另外异或运算还能实现不通过临时变量，就能交换两个变量的值：

```
int a = 5, b = 7;
a = a ^ b;
b = b ^ a;
a = a ^ b;
```

1.3.4 按位非 ~

按位非运算符 ~ 是单目运算符，其功能是将操作数中的二进制位 0 变成 1，1 变成 0。

1.3.5 左移 <<

表达式 a << b 的值是将 a 各二进制位全部左移 b 位后得到的值。左移时，高位丢弃，低位补 0。注意 << 是双目运算符，这里是一个表达式，a 的值不因运算而改变。

实际上，左移 1 位，就等于是乘 2，左移 n 位，就等于乘 2^n。而左移操作比乘法操作快得多。

1.3.6 右移 >>

表达式 a >> b 的值是：将 a 各二进制位全部右移 b 位后得到的值。右移时，移出最右边的位就被丢弃。同样地，a 的值不因右移运算而改变。

对于无符号数，右移时高位补 0。对于有符号数，如 long、int、short、char 类型变量，在右移时，符号位（即最高位）将一起移动，并且大多数 C/C++ 编译器规定，如果原符号位为 1，则右移时最高位就补充 1，原符号位为 0，则右移时高位就补充 0。

实际上，右移 n 位，就相当于左操作数除以 2^n。但是有时候不能除尽，需要将结果往小里取整。

思考题：有两个 int 型的变量 a 和 n（0 <= n <= 31），要求写一个表达式，使该表达式的值和 a 的第 n 位相同。

答案：(a >> n) & 1　　如果 n 不为 31，答案还可以是 (a & (1 << n)) >> n

1.4 引用

1.4.1 引用的定义

定义：类型名 &引用名 = 某变量名;

```
int n = 4;
int &r = n; // r 引用了 n，r 的类型是 int &
```

某个变量的引用，等价于这个变量，相当于该变量的别名。

示例代码：

```
int n = 7;
int &r = n;
r = 4;
cout << r << endl;  // 4
cout << n << endl;  // 4
n = 5;
cout << r << endl;  // 5
```

注意：

- 定义引用时一定要将其初始化成引用某个变量
- 初始化后，它就一直引用该变量，不会再引用别的变量了
- 引用只能引用变量，不能引用常量和表达式

1.4.2 引用的简单示例

交换两个变量的值：

```
void swapByReference(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
int main(int argc, char const *argv[]) {
    int n1 = 3, n2 = 5;
    swapByReference(n1, n2);
```

```
    cout << n1 << " " << n2 << endl;    // 5 3
    return 0;
}
```

引用作为函数的返回值：

```
int n = 4;

int &setValue()
{
    return n;
}

int main(int argc, char const *argv[]) {
    setValue() = 40;
    cout << n << endl;
    return 0;
}
```

常引用：

```
int n = 100;
const int &r = n;
// r = 200; // error: cannot assign to variable 'r' with const-qualified type 'const int &'
n = 300;    // OK
cout << n << endl;  // 300
return 0;
```

`const T &` 和 `T &` 是不同的类型，分别是常引用和非常引用。
常引用和非常引用的转换：

- `const T` 类型的引用或 `T` 类型的变量可以用来初始化 `const T &` 类型的引用。
- `const T` 类型的常变量和 `const T &` 类型的引用则不能用来初始化 `T &` 类型的引用，除非进行强制类型转换。

1.5 const 关键字的用法

1.5.1 定义常量

```
const int MAX_VAL = 23;
const double PI = 3.14;
const char *SCHOOL_NAME = "Ocean University of China";
```

#define 也可以用来定义常量，但应该多使用 `const`，少用 *#define*，因为 `const` 是有类型的，便于类型检查。

1.5.2 定义常量指针

在声明一个指针的时候，在其类型前面加上 const 限定符，这个指针就成为常量指针。

当我们通过常量指针来访问它所指向的内容的时候，不管它指向的内容实际上是否为常量，常量指针都认为它是常量。因此，不可以通过常量指针修改其指向的内容：

```
int n, m;
const int *p = &n;
// *p = 5;  // error: read-only variable is not assignable
n = 4;      // OK
p = &m;     // OK
```

不能把常量指针赋给非常量指针（除非强制类型转换），但反过来可以。因为常量指针认为自己指向的内容是常量，神圣不容修改，如果把这些内容的地址交给非常量指针，内容就有可能被修改，这太危险了，坚决不能给。非常量的指针指向的内容是可以被修改的，非常量指针要把这些内容给常量指针，常量指针就会说“可以啊，就算把地址给我也不会去修改它的”。

```
const int *p1;
int *p2;
p1 = p2;    // OK
p2 = p1;    // error: assigning to 'int *' from incompatible type 'const int *'
p2 = (int *)p1; // OK, cast type
```

由于这个指针本身不是常量（指针常量），所以这个指针可以重新指向其他的内容：

```
int value1 = 5;
const int *ptr = &value1;   // ptr points to a const int
int value2 = 6;
ptr = &value2;  // OK, ptr now points at some other const int
```

总而言之，不允许非常量指针指向常量，也不允许通过常量指针来修改它所指向的内容。

函数参数为常量指针时，可避免函数内部不小心改变参数指针所指地方的内容：

```
void myPrintf(const char *p)
{
    // error: no matching function for call to 'strcpy'
    // note: candidate function not viable: 1st argument ('const char *') would lose
    // const qualifier char    *strcpy(char *__dst, const char *__src);
    // strcpy(p, "this");
    cout << p << endl;
}
```

1.5.3 定义指针常量

我们也可以把某个指针本身定义为常量，称为指针常量。一个指针的值在初始化后就不能被改变，这个指针就称为指针常量。

```
int value = 5;
int *const ptr = &value;
```

和普通的常量一样，指针常量在声明的时候就要进行初始化为某个值。这就意味着，指针常量总是指向相同的地址。

```
int value1 = 5;
int value2 = 6;
// OK, the const pointer is initialized to the address of value1
int *const ptr = &value1;
// not OK, a const pointer can not be changed
ptr = &value2;
```

如果指针常量指向的是变量而不是常量，就可以通过指针常量来修改该变量的值。

```
int value = 5;
int *const ptr = &value;    // ptr will always point to value
*ptr = 6;   // allowed, since ptr points to a non-const int
```

可以同时声明一个指针既是常量指针也是指针常量。这时，既不能修改这个指针常量本身的值，也不能通过它修改它所指向的内容。

```
int value = 5;
const int *const ptr = &value;
```

1.5.4 定义常引用

不能通过常引用修改其引用的变量：

```
int n;
const int &r = n;
// r = 5;  // error: cannot assign to variable 'r' with const-qualified type 'const int &'
```

注意：被引用的变量的值可以通过其他方式改变。

1.6 动态内存分配

1.6.1 用 new 运算符实现动态内存分配

第一种用法，分配一个变量：

```
P = new T;
```

其中，T 是任意类型名，P 是类型为 T * 的指针。动态分配出一片大小为 sizeof(T) 字节的内存空间，并且将该内存空间的起始地址赋值给 P。比如：

```
int *pn;
pn = new int;
*pn = 5;
```

第二种用法，分配一个数组：

```
P = new T[N];
```

其中，T 是任意类型名，P 是类型为 T * 的指针，N 为要分配的数组元素的个数，可以是整型表达式。动态分配出一片大小为 N * sizeof(T) 字节的内存空间，并且将该内存空间的起始地址赋值给 P。

动态分配数组示例：

```
int *pn;
int i =5;
pn = new int[i * 20];
pn[0] = 20;
pn[100] = 30;   // 编译没错，运行时有可能导致数组越界报错。gcc/g++ 不检查数组越界。
return 0;
```

new T 和 new T[n] 这两个表达式的返回值类型都是 T *。

1.6.2 用 delete 运算符释放动态分配的内存

用 new 动态分配的内存空间，一定要用 delete 运算符进行释放：

```
delete pointer;  // pointer 必须指向 new 出来的空间
```

一片动态分配的内存空间不能被 delete 两次，否则运行时会报出异常。

用 delete 释放动态分配的数组时，要加上 []：

```
delete[] pointer;  // pointer 必须指向 new 出来的数组
```

1.7 内联函数、函数重载和函数缺省参数

1.7.1 内联函数

函数调用是有时间开销的。如果函数本身只有几条语句，执行非常快，而且函数被反复执行很多次，相比之下调用函数所产生的这个开销就会显得比较大。

为了减少哦函数调用的开销，引入了内联函数机制。编译器处理对内联函数的调用语句时，是将整个函数的代码插入到调用语句处，而不会产生调用函数的语句。

在函数定义前加 inline 关键字，即可定义内联函数：

```
inline int max(int a, int b)
{
    if (a > b) {
        return a;
    }
    return b;
}
```

以上面的代码为例，max 函数本身就比较简单，如果它被反复多次执行的话，调用 max 函数的产生的额外开销和 max 函数本身执行时的开销是相当的，所以可以将其定义为内联函数来减少调用它的时间

开销。通过 inline 声明，编译器首先在函数调用处使用函数本身语句替换了函数调用语句，然后编译替换后的代码，运行时不需要跳转到内存其他地址去执行函数调用，也不需要保留函数调用时的现场数据。通过将函数声明为内联，可以将函数的定义放在头文件内，在类内部定义的函数会默认声明为内联函数。

但是由于代码的扩展，内联函数可能会增大可执行程序的体积；如果内联函数发生了改动，需要重新编译代码。而内联声明只是对编译器的一种建议，编译器会自己决定是否采用内联措施，这取决于函数是否符合内联的有利条件。如果函数体非常大，编译器将会忽略内联声明，将内联函数作为普通函数来处理。

1.7.2 函数重载

一个或多个函数，名字相同，然而参数个数或参数类型不相同，这叫做函数的重载。

以下三个函数是重载关系：

```
int Max(double f1, double f2);
int Max(int n1, int n2);
int Max(int n1, int n2, int n3);
```

函数重载使得函数命名变得简单。编译器根据调用语句中的实参个数和类型判断应该调用哪个函数，有时会出现二义性的错误。C 语言里是没有函数重载的，函数命名不能相同。

```
Max(3.4, 2.5);  // 调用 int Max(double f1, double f2)
Max(2, 4);      // 调用 int Max(int n1, int n2)
Max(1, 2, 3);   // 调用 int Max(int n1, int n2, int n3)
Max(3, 2.4);    // error: call to 'Max' is ambiguous
```

如果两个函数函数名和参数都一样，只是返回值不同，这时不能重载，编译时会报错：

```
int Max(double f1, double f2) {}
// error: functions that differ only in their returntype cannot be overloaded
// double Max(double f1, double f2) {}
```

1.7.3 函数的缺省参数

C++ 中，定义函数的时候可以让最右边的连续若干个参数有缺省值，那么调用函数的时候，若相应位置不写参数，参数就是缺省值。

```
void func(int x1, int x2 = 2, int x3 = 3) {}
// error: missing default argument on parameter 'x2'
// void func(int x1 = 1, int x2, int x3 = 3) {}
// when call the function func
func(10);       // OK, same as func(10, 2, 3);
func(10, 8);    // OK, same as func(10, 8, 3);
// func(10, , 8);  // error. 只能最右边的连续若干个参数缺省
```

函数可缺省的目的在于提高程序的可扩充性。即如果某个写好的函数要添加新的参数，而原来那些调用该函数的语句，未必需要使用新增的参数，那么为了避免对原先那些函数调用语句的修改，就可以使用缺省参数。

2 浅谈面向对象的程序设计

2.1 面向对象的基本概念

结构化程序设计：

- 复杂的大问题 → 层层分解/模块化 → 若干子问题
- 自顶向下，逐步求精

程序 = 数据结构（变量）+ 算法（函数）

在结构化程序设计中，数据结构与算法没有直观的联系，会遇到四大问题：理解难、修改难、查错难、重用难。

软件设计的目标是更快、更正确、更经济，所以业界需要面向对象。

面向对象的程序设计方法继承了结构化程序设计的优点，同时又克服了结构化程序设计中的一些不足，它的设计思路更加接近于我们的现实生活。

面向对象的程序 = 类 + 类 + ... + 类

面向对象的程序设计中的基本概念：

抽象 归纳出一类事物的共同属性（数据结构）和行为（函数）

封装 将数据结构和算法对应地捆绑在一起

继承

多态

2.2 面向对象的程序设计语言的发展历程

早起程序设计语言的历史：

- ALGOL 60: 1960 年
- CPL: 1963 年
- BCPL: 1967 年
- B: 1970 年
- C: 1973 年

面向对象的程序设计语言的历史：

- Simula 67: 1967 年
- Smalltalk: 1971 年
- C++: 1983 年
- Java: 1995 年
- C#: 2003 年

C++ 标准的发展：

- C++2.0: 1989 年
- ANSI C++: 1994 年
- C++98: 1998 年，加入 STL（Standard Template Library）泛型设计
- C++03: 2003 年
- *Libraray Technical Report 1*(TR1): 2005 年
- C++11: 2011 年

常用的 C++ 编译器

- GCC
- Visual C++
- Dev C++
- Eclipse
- Borland C++Builder

3 类和对象

3.1 类的定义

类包含成员变量和成员函数，二者统称为类的成员。可以将类看作是带有函数的结构体。
类的定义：

```
class 类名 {
    访问范围说明符:
        成员变量1
        成员变量2
        ...
        成员函数声明1
        成员函数声明2
    访问范围说明符:
        更多成员变量
        更多成员函数声明
        ...
};
```

以抽象出的矩形类 `CRectangle` 为例：

```
class CRectangle {
    public:
        int w;
        int h;
        void Init(int w_, int h_) {
            w = w_;
            h = h_;
        }
        int Area() {
            return w * h;
        }
        int Perimeter() {
            return 2 * (w + h);
        }
};  // 必须有分号
```

在程序中使用类来创建对象：

```
int main() {
    int w, h;
    CRectangle r;   // r 是一个对象
    cin >> w >> h;
    r.Init(w, h);
    cout << r.Area() << endl << r.Perimeter() << endl;
    return 0;
}
```

类定义的变量 → 类的实例 → “对象”

每个对象都有自己的存储空间，对象的大小是所有成员变量的大小之和。一个对象的某个成员变量被改变，不会影响到其他的对象。

对象之间可以用“=”进行赋值，不能使用“==”、“!=”、“>”、“<”等进行比较，除非这些运算符经过了重载。

3.2 访问类的成员变量和成员函数

3.2.1 访问类的成员变量和成员函数的三种常用方式

用法 1：对象名. 成员名

```
CRectangle r1, r2;
r1.w = 5;
r2.Init(3, 4);
```

用法 2：指针-> 成员名

```
CRectangle r1, r2;
CRectangle *p1 = &r1;
CRectangle *p2 = &r2;
p1->w = 5;
p2->Init(3, 4);
```

用法 3：引用名. 成员名

```
CRectangle r2;
CRectangle &rr = r2;
rr.w = 5;
rr.Init(3, 4);  // rr 的值变了，r2 的值也变了
```

另一种输出结果的方式：

```
void printRectangle(CRectangle &r) {
    cout << r.Area() << ", " << r.Perimeter();
}
CRectangle r3;
r3.Init(3, 4);
printRectangle(r3);
```

成员函数体和类的定义可以分开写：

```
// CRectangle.h
class CRectangle {
    public:
        int w, h;
        int Area();
        int Perimeter();
        void Init(int w_, int h_);
};

// CRectangle.cpp
int CRectangle::Area() {
    return w * h;
}
int CRectangle::Perimeter() {
    return 2 * (w + h);
}
void CRectangle::Init(int w_, int h_) {
    w = w_;
    h = h_;
}
```

3.2.2 类成员的可访问范围

类成员的可访问范围由以下三种关键字确定：

- **private**: 指定私有成员，只能在成员函数内被访问
- **public**: 指定公有成员，可以在任何地方被访问
- **protected**: 指定保护成员（后面讲“继承”时会讲到）

三种关键字出现的次数和先后次序都没有限制。如果没有指定，默认为私有成员（**private**）。

```
class Man {
        int nAge;           // private
        char szName[20];    // private
    public:
        void SetName(char *name) {
            strcpy(szName, name);   // OK
        }
}
```

类的成员函数内部，可以访问：

- 当前对象的全部属性、函数
- 同类其它对象的全部属性、函数

类的成员函数以外的地方：

- 只能访问该类对象的公有成员

设置私有成员的目的是强制对成员变量的访问一定要通过成员函数进行，从而有效地将私有成员隐藏起来。

3.3 内联成员函数和重载成员函数

3.3.1 内联成员函数

前面已经学到了内联函数。它是指一类特殊的函数，其自身的语句非常的少，运行起来也非常快，在这种情况下，由于函数调用会产生一定的调用语句，那么当函数需要被反复调用时，就会产生非常大的调用开销。如果将其定义为内联函数，编译器会将此函数的定义插入到调用的地方，就不会产生额外的调用开销了。成员函数也可以被定义为内联函数，称为内联成员函数。

定义内联成员函数的两种方式：

- **inline** + 成员函数
- 整个函数体出现在类定义内部

```
class A {
    inline void func1();
```

```
    void func2() {
        // definition
    }
};

void A::func1() {}
```

3.3.2 重载成员函数

类的成员函数也可以重载、带缺省参数。

两个重载函数的返回类型可以不一样，如 `int func(int x) {}` 和 `void func(){}` 是重载关系。但是如果两个函数仅在返回类型上有区别，那就会报重复定义的错误。

缺省参数既可以在函数声明时定义，也可以在函数实现时定义，但二者不能同时定义。

```
class Location {
private:
    int x;
    int y;
public:
     void init(int x = 0, int y = 0);   // 可以在声明或实现时定义默认参数，但不能二者同时定义
     void value_x(int val) {x = val;}
     int value_x() {return x;}          // 重载函数返回类型可以不一样
};

void Location::init(int X, int Y) {
    x = X;
    y = Y;
}

int main(int argc, char const *argv[]) {
    Location A;
    A.init(5);
    A.value_x(5);
    cout << A.value_x() << endl;
    A.value_x();
    cout << A.value_x() << endl;
    return 0;
}
```

3.4 构造函数

3.4.1 构造函数的基本概念

构造函数是成员函数的一种，名字与类名相同，可以有参数，但不能有返回值（void 也不行）。构造函数的作用是对对象进行初始化，如给成员变量赋初值。

如果定义类时没写构造函数，则编译器生成一个默认的无参数的构造函数，不做任何操作。如果定义了构造函数，则编译器不生成默认的无参构造函数。一个类可以定义多个构造函数。对象生成时构造函数自动被调用。对象一旦生成，就再也不能在其上执行构造函数。

为什么需要构造函数：

1. 构造函数执行必要的初始化工作，有了构造函数就不用专门再写初始化函数，也不用担心忘记调用初始化函数。
2. 有时对象没被初始化就使用，会导致程序出错。

例如，定义了一个类 Complex，但没有定义构造函数。在生成对象时，会调用默认的构造函数：

```
class Complex {
private:
    double real, imag;
public:
    void set(double r, double i);
};  // 编译器自动生成默认构造函数

int main(int argc, char const *argv[]) {
    Complex c1;                  // 默认构造函数被调用
    Complex *pc = new Complex;  // 默认构造函数被调用
    return 0;
}
```

如果定义了构造函数，编译器就不会生成默认的无参构造函数了：

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r, double i = 0);     // 构造函数也需要声明
    void set(double r, double i);
};

Complex::Complex(double r, double i) {
    real = r;
    imag = i;
}
```

```
int main(int argc, char const *argv[]) {
    // error: no matching constructor for initialization of 'Complex'
    // Complex c1;
    // error: no matching constructor for initialization of 'Complex'
    // Complex *pc = new Complex;
    Complex c1(2);                      // OK
    Complex c2(2, 4), c3(3, 5);         // OK
    Complex *pc = new Complex(3, 4);    // OK
    return 0;
}
```

一个类可以有多个构造函数，多个构造函数之间是重载的关系。
构造函数最好是 public 的，private 构造函数不能直接用来初始化对象。

3.4.2 构造函数在数组中的使用

```
class CSample {
private:
    int x;
public:
    CSample() {
        cout << "CSample constructor 1 is called." << endl;
    }
    CSample(int n) {
        x = n;
        cout << "CSample constructor 2 is called." << endl;
    }
};

class Test {
public:
    Test(int n) {
        cout << "Test constructor 1 is called." << endl;
    }
    Test(int n, int m) {
        cout << "Test constructor 2 is called." << endl;
    }
    Test() {
        cout << "Test constructor 3 is called." << endl;
    }
};
```

```
int main(int argc, char const *argv[]) {
    CSample array1[2];  // 对象数组
    // CSample constructor 1 is called.
    // CSample constructor 1 is called.
    cout << "step 1" << endl;
    CSample array2[2] = {4, 5}; // 等同于 CSample array2[2] = {CSample(4), CSample(5)};
    // CSample constructor 2 is called.
    // CSample constructor 2 is called.
    cout << "step 2" << endl;
    CSample array3[2] = {3};
    // CSample constructor 2 is called.
    // CSample constructor 1 is called.
    cout << "step 3" << endl;
    CSample *array4 = new CSample[2];
    // CSample constructor 1 is called.
    // CSample constructor 1 is called;
    delete[] array4;

    Test testArray1[3] = {1, Test(1, 2)};
    // Test constructor 1 is called
    // Test constructor 2 is called
    // Test constructor 3 is called
    Test testArray2[3] = {Test(2, 3), Test(1, 2), 1};
    // Test constructor 2 is called
    // Test constructor 2 is called
    // Test constructor 1 is called
    Test *pArray[3] = {new Test(4), new Test(1, 2)}; // pArray[3] 是指针数组，不是对象数组
    // Test constructor 1 is called
    // Test constructor 2 is called
    // pArray[2] 是一个未初始化的指针
    return 0;
}
```

3.4.3 复制（拷贝）构造函数

复制构造函数只有一个参数，即对同类对象的引用。形如 `X::X(X &)` 或 `X::X(const X &)` 二者中的一种，后者能以常量对象作为参数。

如果没有定义复制构造函数，那么编译器会生成默认的复制构造函数。

```
class Complex {
private:
    double real;
    double imag;
```

```
};

int main(int argc, char const *argv[]) {
    Complex c1;     // 调用缺省无参构造函数
    Complex c2(c1); // 调用缺省的复制构造函数，将 c2 初始化成和 c1 一样
    return 0;
}
```

如果定义自己的复制构造函数，那么默认的复制构造函数将不存在，因为复制构造函数只能有一个。

```
class Complex {
private:
    double real;
    double imag;
public:
    Complex() {}    // 复制构造函数也是构造函数，一旦定义了，默认的无参构造函数会消失
    Complex(const Complex &c) {
        real = c.real;
        imag = c.imag;
        cout << "Complex copy constructor is called." << endl;
    }
};

int main(int argc, char const *argv[]) {
    Complex c1;     // 调用自定义无参构造函数
    Complex c2(c1); // 调用自定义的复制构造函数
    // Complex copy constructor is called.
    return 0;
}
```

不允许有形如 `X::X(X)` 的复制构造函数，它的参数应该是引用，而不是对象:

```
class CSample {
    CSample(CSample c) {
    }   // 错误，不允许这样的构造函数
};
```

复制构造函数起作用的三种情况:

1. 当用一个对象去初始化同类的另一个对象时。

```
Complex c2(c1);
Complex c2 = c1;    // 同上，初始化语句，非赋值语句
```

2. 如果函数有一个参数是类 A 的对象，那么该函数被调用时，类 A 的复制构造函数将被调用。

```
class A {
public:
    A() {};
    A(A &a) {
        cout << "A copy constructor is called." << endl;
    }
};
void Func1(A a1) {}
int main() {
    A a2;
    Func1(a2);   // 由于是自定义复制构造函数，实参 a2 和形参 a1 不一定相等
    // A copy constructor is called.
    return 0;
}
```

3. 如果函数的返回值是类 A 的对象时，则函数返回时，A 的复制构造函数被调用。

```
class B {
public:
    int v;
    B(int n) {v = n;}
    B(const B &a) {
        v = 5;
        cout << "B copy constructor is called." << endl;
    }
};

B Func2() {
    B b(4);
    return b;
}

int main(int argc, char const *argv[]) {
    cout << Func2().v << endl;
    // B copy constructor is called.
    // 5
    return 0;
}
```

注意：某些编译器如 G++ 会针对返回值进行优化，导致返回对象时不会调用复制构造函数，编译时可以加上 `-fno-elide-constructors` 参数来关闭这项优化。[1]

[1] 《C++ 返回值为对象时复制构造函数不执行怎么破》https://www.kancloud.cn/digest/bugkiller/163413

3.4.4 类型转换构造函数

对于一个类 X 来说，当它的某个构造函数只有一个参数，且该参数又不是 const X & 类型时，这个构造函数就称为类型转换构造函数。

类型转换构造函数的目的是将一个其他类型的数据转换成一个类的对象。类型转换构造函数只有一个参数 (C++11 之前)，不是复制构造函数。

```
class Complex {
public:
    double real, imag;
    Complex(int i) {    // 类型转换构造函数
        cout << "Int constructor of Complex is called." << endl;
        real = i;
        imag = 0;
    }
    Complex(double r, double i) {
        cout << "Constructor of Complex is called." << endl;
        real = r;
        imag = i;
    }
};

int main(int argc, char const *argv[]) {
    Complex c1(7, 8);
    // Constructor of Complex is called.
    Complex c2 = 12;    // 注意这里的 = 是初始化，不是赋值
    // Int constructor of Complex is called.
    c1 = 9; // OK. 9 被自动转换成一个临时的 Complex 对象，然后赋值给 c1
    // Int constructor of Complex is called.
    return 0;
}
```

3.5 析构函数

3.5.1 析构函数的定义

析构函数也是成员函数的一种，负责处理消亡的对象。在对象消亡时，析构函数会自动被调用，做一些如释放分配的内存空间等善后工作。

析构函数的函数名与类名相同，不过函数名前面有个 ~，没有参数和返回值。一个类最多有一个析构函数。

定义类时如果没有写析构函数，编译器会生成缺省析构函数。缺省析构函数不涉及释放用户申请的内存释放等清理工作。如果定义了析构函数，则编译器不生成缺省析构函数。

```
class String {
```

```
private:
    char *p;
public:
    String() {
        p = new char[10];
    }
    ~String();
};

String::~String() {
    delete[] p;
}
```

3.5.2 析构函数和数组

当对象数组的生命周期结束时，对象数组的每个元素都会调用到析构函数。

```
class Ctest {
public:
    ~Ctest() {
        cout << "Destructor of Ctest is called" << endl;
    }
};

int main(int argc, char const *argv[]) {
    Ctest array[2];
    cout << "End main" << endl;
    // End main
    // Destructor of Ctest is called
    // Destructor of Ctest is called
    return 0;
}
```

3.5.3 析构函数和运算符 delete

用 new 运算新建对象会调用构造函数，类似地，用 delete 运算释放对象会调用析构函数。

```
Ctest *pTest;
pTest = new Ctest;  // 构造函数调用
delete pTest;       // 析构函数调用
// Destructor of Ctest is called
```

但是，如果用 new 新建的对象只能用 delete 释放，不会自动调用析构函数。

```
class A {
public:
    A() {
        cout << "Constructor of A is called." << endl;
    }
    ~A() {
        cout << "Destructor of A is called." << endl;
    }
};

int main(int argc, char const *argv[]) {
    A *p = new A[2];
    // Constructor of A is called.
    // Constructor of A is called.
    A *p2 = new A;  // 用 new 创建的对象不会自动释放，不会调用析构函数
    // Constructor of A is called.
    A a;
    // Constructor of A is called.
    delete[] p;
    // Destructor of A is called.
    // Destructor of A is called.
    // Destructor of A is called.
    return 0;
}
```

3.6 静态成员变量和静态成员函数

静态成员：在声明前面加了 **static** 关键字的成员。

```
class CRectangle {
private:
    int w, h;
    static int nTotalArea;      // declare static member variable nTotalArea
    static int nTotalNumber;    // declare static member variable nTotalNumber
public:
    CRectangle(int w_, int h_);
    ~CRectangle();
    static void PrintTotal();   // declare static member function PrintTotal()
};
```

静态成员属于整个类，不属于某些特定的对象。

普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。**sizeof** 运算符计算一个对象所占的内存空间时，不会计算静态成员变量。也就是说，静态成员变量其实不是存储在

对象内部的。

普通成员函数必须具体作用于某个对象，而静态成员函数并不具体作用于某个对象，不需要通过对象就能访问。

访问静态成员的方式（前提是有访问权限）：

1. 类名::成员名（首选方式）

```
CRectangle::PrintTotal();
```

2. 对象名.成员名

```
CRectangle r;
r.PrintTotal();
```

3. 指针->成员名

```
CRectangle *p = &r;
p->PrintTotal();
```

4. 引用.成员名

```
CRectangle &ref = r;
int n = ref.nTotalNumber;
```

静态成员变量本质上是全局变量，哪怕一个对象都不存在，类的静态成员变量也存在。静态成员函数本质上是全局函数。

设置静态成员的这种机制的目的是将和某些类紧密相关的全局变量和函数写到类里面，看上去像一个整体，易于理解和维护。

类里面涉及静态成员变量的语句仅仅是对静态成员变量的声明，不是定义，定义时不需要加 `static` 关键字。在使用静态成员变量的时候，需要在所有函数外面对静态成员变量进行定义，而且不管有没有访问权限，都可以选择是否对其进行初始化。静态变量的初始化必须在全局的范围，而且必须放置在类的定义之后，否则会产生 `error: incomplete type 'ClassName' named in nested name specifier` 的错误。

```
// 定义静态变量，并初始化为 0（只在声明时加 static 关键字，定义时不加）
int CRectangle::nTotalNumber = 0;
int CRectangle::nTotalArea = 0;
```

为什么要这样设计呢？为什么不在类内定义它呢？这是由于遵循了每个要使用的静态对象都要定义且仅定义一次的原则。类的定义一般都会放在头文件（.h）里，但是静态变量的使用一般在源程序文件（.cpp）里，要使用就要在源程序文件内对其定义。而如果定义类的时候也定义静态变量，这就有可能导致多次定义的错误。[2]

在静态成员函数中，不能访问非静态成员变量，也不能调用非静态成员函数。这是因为静态成员函数不是作用在某个特定的对象上的，非静态成员变量属于特定的对象。但是反过来，在非静态成员函数中是可以调用静态成员函数的。

[2] Why does a static data member need to be defined outside of the class? https://stackoverflow.com/questions/18749071/why-does-a-static-data-member-need-to-be-defined-outside-of-the-class

静态成员变量可以被派生类继承，但是基类和派生类的静态成员变量共用一份内存。不能通过成员初始化列表来初始化静态成员变量，也不能通过派生类来初始化其从基类继承的静态成员变量。

静态成员变量是可以随时被赋值的，但是如果加上 `const` 限定符后，静态成员变量就会成为静态成员常量，仅能在全局范围进行初始化定义，在其他任何地方都不能被赋值。

3.7 成员对象、封闭类和成员初始化列表

3.7.1 成员对象和封闭类的定义

成员对象：一个类的成员变量是另一个类的对象，这种成员变量就称为成员对象。

封闭类：包含成员对象的类。

```
class CTyre {
private:
    int radius;
    int width;
public:
    // Member initializer lists. radius = r; width = w;
    CTyre(int r, int w) : radius(r), width(w) {}
};

class CEngine {
};

class CCar {    // enclosing class
private:
    int price;
    CTyre tyre;
    CEngine engine;
public:
    CCar(int p, int tr, int tw);
};
```

3.7.2 成员初始化列表

前面 `CTyre(int r, int w):radius(r), width(w) {}` 中用到了成员初始化列表。在介绍成员初始化列表之前，先补充一下 C++ 中的三种初始化变量的方式（注意是初始化，不是赋值）：

1. 复制初始化

```
int nValue = 5; // copy initialization
```

2. 直接初始化

```
int nValue(5);  // direct initialization
```

3. 统一初始化（C++11）

```
int nValue{5};  // uniform initialization or brace initialization (C++11)
```

之前我们在写构造函数的时候，是利用赋值运算符通过赋值的方式进行的复制初始化。这不是推荐的方式，有些时候会比较低效。而且对于某些数据类型，例如 const 和引用，必须在声明的时候就要初始化：

```
class Something {
private:
    const int m_value;
public:
    Something() {
        // error: constructor for 'Something' must explicitly initialize the const member
        m_value = 1;
    }
};

const int m_value; // error: const vars must be initialized with a value
m_value = 5; //  error: const vars can not be assigned to
```

而我们有些时候并不想在声明类的成员常量时就指定一个值，而是想在创建对象并初始化的时候再指定。

如果在前面的封闭类 CCar 中不定义构造函数，让编译器使用默认的构造函数。创建对象 car 并初始化时，对于成员对象 car.engine，编译器可以使用默认构造函数初始化，对于没有默认构造函数的成员对象 car.tyre 就不知道该如何初始化了，所以就会报错。

成员初始化列表可以解决上面的这些问题。

定义构造函数时，添加成员初始化列表：

```
类名::构造函数(参数表) : 成员变量1(参数表), 成员变量2(参数表), ... // 直接初始化成员变量
{
    // 不需要在这里写赋值语句
}
```

成员对象初始化列表中的参数：

- 任意复杂的表达式
- 函数/变量/表达式中的函数，变量必须有定义

利用成员初始化列表来初始化成员常量：

```
class Something {
private:
    const int m_value;

public:
```

```
    Something(): m_value(5) {} // directly initialize our const member variable
};
```

注意：静态成员变量不能通过成员初始化列表进行初始化。不管有没有访问权限，静态变量都和全局变量类似，需要在全局范围内进行初始化定义。

3.7.3 封闭类的调用顺序

封闭类的对象遵循“先构造的后析构”的顺序。
当封闭类对象生成时：

1. 执行所有成员对象的构造函数
2. 执行封闭类的构造函数

当封闭类的对象消亡时：

1. 执行封闭类的析构函数
2. 执行成员对象的析构函数

成员对象的构造函数调用顺序与成员对象在类中的声明一致，与其在成员初始化列表中出现的顺序无关。

3.8 友元

3.8.1 友元函数

一个类的友元函数可以访问该类的私有成员。除此之外，友元函数和其他函数并无差别。友元函数可以是某个普通函数，也可以是某个类的成员函数。声明类的成员函数为友元函数时，要在该函数前加上类名和::。

在一个类中，如果想把哪个函数定义为友元函数，直接写下它的函数原型，并在前面加上一个 **friend** 关键字就可以了，在私有范围还是公有范围都不影响。

```
class Accumulator {
private:
    int m_value;
public:
    Accumulator() { m_value = 0; }
    void add(int value) { m_value += value; }

    // Make the reset() function a friend of this class
    friend void reset(Accumulator &accumulator);
};

// reset() is now a friend of the Accumulator class
void reset(Accumulator &accumulator) {
```

```
    // And can access the private data of Accumulator objects
    accumulator.m_value = 0;
}

int main() {
    Accumulator acc;
    acc.add(5); // add 5 to the accumulator
    reset(acc); // reset the accumulator to 0
    return 0;
}
```

要注意的是，我们必须要把特定的对象传递给友元函数，这样友元函数才能操作其私有成员变量。友元函数无法访问相应类的对象的 *this 指针。一个函数可以同时作为多个类的友元函数。

3.8.2 友元类

可以将一个类声明为另一个类的友元，称为友元类。
A 是 B 的友元类 ⇒ A 的成员函数可以访问 B 的私有成员

```
class CDriver;  // 前置声明

class CCar {
private:
    int price;
    friend class CDriver;   // 声明 CDriver 为友元类
};

class CDriver {
public:
    CCar myCar;
    void ModifyCar() {
        // CDriver 是 CCar 的友元类，可以访问其私有成员
        myCar.price += 1000;
    }
};
```

需要注意，对于一个类来说，其友元类虽然可以访问它的对象的私有成员，但是无法访问它的对象的 *this 指针。A 是 B 的友元类，并不意味着 B 也是 A 的友元类。友元类之间的关系不能传递，不能继承。

一个类的友元函数和友元类破坏了它的封装性。如果这个类的某些细节需要改动，那友元也要跟着做出相应的改动。因此，要尽可能地限制使用友元函数和友元类。

3.9 this 指针

C++ 语言刚诞生的时候是没有能直接将 C++ 语言编译成机器语言的编译器的，而是先翻译成 C 语言，然后再通过 C 的编译器将其编译成机器语言。C++ 语言编译成 C 语言，对象对应结构体，之后就会多一个 this 指针。尽管现在已经有了 C++ 的直接编译器了，但是依然可以这么理解，即会多出一个 this 指针，其作用是指向成员函数所作用的对象。

C++ 代码：

```
class CCar {
public:
    int price;
    void SetPrice(int p);
};
void CCar::SetPrice(int p) {
    price = p;
}
int main() {
    CCar car;
    car.SetPrice(20000);
    return 0;
}
```

翻译后的 C 代码：

```
struct CCar {
    int price;
};
void SetPrice(struct CCar *this, int p) {
    this->price = p;
}
int main() {
    struct CCar car;
    SetPrice(&car, 20000);
    return 0;
}
```

非静态成员函数中可以直接使用 this 来代表指向该函数作用的对象的指针。

```
class Complex {
public:
    double real, imag;
    void Print() {
        std::cout << real << "," << imag << std::endl;
    }
    Complex(double r, double i) : real(r), imag(i) {}
```

```
    Complex AddOne() {
        this->real++;
        this->Print();
        return *this;
    }
};

int main(int argc, char const *argv[]) {
    Complex c1(1, 1), c2(0, 0);
    c2 = c1.AddOne();   // 2,1
    return 0;
}
```

比较下面代码中 A、B 类的不同，体会 this 指针的作用。

```
class A {
private:
    int i;
public:
    void Hello() {
        std::cout << "Hello world!" << std::endl;
    }
};

class B {
private:
    int i;
public:
    void Hello() {
        std::cout << i << " Hello world!" << std::endl;
    }
};

int main(int argc, char const *argv[]) {
    A *p1 = NULL;
    p1->Hello();    // Hello world!
    B *p2 = NULL;
    p2->Hello();    // Segmentation fault: 11
    return 0;
}
```

this 指针并不占用对象的内存空间，不会影响 sizeof 运算符的结果。它与对象并不存在包含关系，只是当对象调用函数时，对象被 this 指针指向。this 指针的存放位置跟编译器有关，可能是栈，也可

能是寄存器，甚至是全局变量。

静态成员函数并不具体作用于某个对象，因此静态成员函数中不能使用 this 指针。普通成员函数的真实参数个数要比参数表中写的多一个 this 指针，而静态成员函数的真实的参数个数就是程序中写出的参数个数。

3.10 常量对象、常量成员函数和常引用

3.10.1 常量对象

如果不希望某个对象的值被改变，则定义该对象的时候可以在前面加上 const 关键字。

```
class Demo {
private:
    int value;
public:
    Demo(int v) : value(v) {}
    void setValue(int v) {
        value = v;
    }
};

int main(int argc, char const *argv[]) {
    const Demo obj(0); // const object

    // error: member function 'setValue' not viable: 'this' argument has type
    // 'const Demo', but function is not marked const
    obj.setValue(1);
    return 0;
}
```

常量对象一旦通过构造函数完成了初始化，它的成员变量就不允许通过任何方式修改，不管是直接修改还是通过成员函数修改都不行。常量对象只可以调用其常量成员函数。

3.10.2 常量成员函数

在类的成员函数声明后面可以加 const 关键字，则该成员函数成为常量成员函数。如果该常量成员函数需要在类外定义，则类内的声明后面和类外的定义后面都要加 const 关键字。

常量成员函数执行期间不应该修改其作用的对象。因此，在常量成员函数中可以访问但不能修改成员变量的值（静态成员变量除外），也不能调用同类的非常量成员函数（静态函数成员除外）。

```
class Sample {
public:
    int value;
    Sample() {}
    void getValue() const;
```

```
    void func() {}
};

void Sample::getValue() const { // here need 'const' keyword in definition
    // error: cannot assign to non-static data member within const member
    // function 'getValue'
    value = 0;
    // error: member function 'func' not viable: 'this' argument has type
    // 'const Sample', but function is not marked const
    func();
}

int main() {
    const Sample o; // 大部分编译器需要为 Sample 类编写构造函数，然后才能
    // error: cannot assign to variable 'o' with const-qualified type
    // 'const Sample'
    o.value = 100;
    // error: member function 'func' not viable: 'this' argument has type
    // 'const Sample', but function is not marked const
    o.func();
    o.getValue();   // OK
    return 0;
}
```

两个成员函数，名字和参数表都一样，但是一个是 const，一个不是，它们是重载的关系，不是重复定义。

3.10.3 常引用

引用前面可以加 const 关键字，成为常引用。不能通过常引用修改其引用的变量。

```
class Sample {
    ...
};
void printfObj(const Sample &o) {
    ...
}
```

4 运算符重载

4.1 运算符重载的基本概念

C++ 中预定义的运算符如 +、-、*、/ 等，只能用于如整型、实型、字符型、逻辑型等基本的数据类型的运算。

C++ 提供了数据抽象的手段，允许用户通过类自定义数据类型，然后通过调用类的成员函数来操作它的对象。使用类的成员函数来操作对象很不方便，我们希望像数学上一样，使用运算符来操作不同的对象，这时就可以对运算符进行重载。

运算符重载是指对已有的运算符赋予多重的含义，使同一运算符作用于不同类型的数据时产生不同的行为。运算符重载的实质是函数重载。

```
返回值类型 operator 运算符(形参表)
{
    ...
}
```

运算符重载的目的是扩展 C++ 中提供的运算符的适用范围，以用于类所表示的抽象数据类型。运算符重载使得抽象数据类型也能够直接使用 C++ 提供的运算符，使程序更简洁、代码更容易理解。

在程序编译时：

- 把含运算符的表达式 → 对运算符函数的调用
- 把操作符的操作数 → 运算符函数的参数
- 运算符被多次重载时，根据实参的类型决定调用哪个运算符函数
- 运算符可以被重载为普通函数，也可以被重载为成员函数

4.1.1 运算符被重载为普通函数

```
class ComplexN {
public:
    ComplexN(double r = 0.0, double i = 0.0) {
        real = r;
        imaginary = i;
    }
    double real;
    double imaginary;
};

ComplexN operator+(const ComplexN &a, const ComplexN &b) {
    return ComplexN(a.real + b.real, a.imaginary + b.imaginary);
}
int main(int argc, char const *argv[]) {
    ComplexN a(1, 2), b(2, 3), c;
    c = a + b;
    return 0;
}
```

运算符被重载为普通函数时，参数个数为运算符目数。

4.1.2 运算符被重载为成员函数

```
class ComplexM {
public:
    ComplexM(double r = 0.0, double m = 0.0) : real(r), imaginary(m) {}
    ComplexM operator+(const ComplexM &);
    ComplexM operator-(const ComplexM &);
private:
    double real;
    double imaginary;
};

ComplexM ComplexM::operator+(const ComplexM &operand) {
    return ComplexM(real + operand.real, imaginary + operand.imaginary);
}

ComplexM ComplexM::operator-(const ComplexM & operand) {
    return ComplexM(real - operand.real, imaginary - operand.imaginary);
}

int main(int argc, char const *argv[]) {
    ComplexM x, y(4.3, 8.2), z(3.3, 1.1);
    x = y + z;  // same as x = y.operator+(z);
    x = y - z;  // same as x = y.operator-(z);
    return 0;
}
```

运算符被重载为成员函数时，参数个数为运算符目数减一。

4.2 赋值运算符的重载

如果赋值运算符 = 两边的类型不匹配，我们就需要重载赋值运算符 = 。赋值运算符 = 只能重载为成员函数，不能重载为普通函数。

4.2.1 赋值运算符重载实例

下面的代码编写了一个长度可变的字符串类 MyString，其成员包含一个 char * 类型的成员变量，指向动态分配的存储空间，该存储空间用于存放 '\0' 结尾的字符串。

```
class MyString {
private:
    char *str;
public:
    MyString() : str(NULL) {}
```

```
    const char *c_str() {return str;}
    char *operator=(const char *s);
    ~MyString();
};

char *MyString::operator=(const char *s) {
    if (str) delete[] str;
    if (s) {    // s is not NULL
        str = new char[strlen(s) + 1];  // + 1 for '\0'
        strcpy(str, s);
    } else {
        str = NULL;
    }
    return str;
}

MyString::~MyString() {
    if (str) delete[] str;
}

int main() {
    // no constructor MyString(char *)
    // error: no viable conversion from 'const char [11]' to 'MyString'
    // MyString s = "Good luck!";
    MyString s;
    s = "Good luck!";   // same as s.operator=("Good luck!");
    std::cout << s.c_str() << std::endl;    // Good luck!
    s = "Ocean University of China";
    std::cout << s.c_str() << std::endl;    // Ocean University of China

    MyString s1, s2;
    s1 = "this";
    s2 = "that";
    // Note: here won't call char *operator=(const char *s); the types don't match!
    // So it will call the default operator= by the compiler, use shallow copy
    s1 = s2;    // runtime error:  pointer being freed was not allocated
}
```

4.2.2 重载赋值运算符的意义——浅拷贝和深拷贝

C++ 并不了解我们自定义的类，它会为我们的类自动重载赋值运算符，采用的是方式是浅拷贝（shallow copy, field-by-field copy or memberwise copy），这种赋值运算符的原型如下，它接受并返回一个指

向类对象的引用：

```
ClassName &ClassName::operator=(const ClassName &);
```

C++ 默认的复制构造函数和默认重载的赋值运算符都执行是浅拷贝。

对一个对象的浅拷贝会拷贝所有它所有成员的值。例如，如果该对象有一个 `int` 类型的成员，那么就把 `int` 类型的值拷贝过来。但是，如果该对象包含一个指针类型的成员，该指针指向一块动态申请的内存，这时就会出现问题了。因为浅拷贝只会把这个指针成员拷贝一份，保证两个指针的值一样，但不会拷贝该指针所指向的内存空间。因此，如果一个对象的成员里包含指针，浅拷贝之后，两个对象的指针成员会指向同一块内存。

而对一个对象进行深拷贝则不同，它会将该对象中指针成员指向的内容也复制一份到另一个对象指针成员指向的地方。深拷贝后，两个对象的指针成员分别指向两块不同的内存空间，但这两块内存空间的内容是一样的。

重载赋值运算符正确的写法：

```
MyString &MyString::operator=(const MyString &s) {
    if (str == s.str) return *this; // check self-assignment
    if (str) delete[] str;
    str = new char[strlen(s.str) + 1];
    strcpy(str, s.str);
    return *this;   // to chain this operator, eg x = y = z
}
```

复制构造函数正确的写法：

```
MyString::MyString(const MyString &s) {
    if (s.str) {    // s is not NULL
        str = new char[strlen(s.str) + 1];  // + 1 for '\0'
        strcpy(str, s.str);
    } else {
        str = NULL;
    }
}
```

注意：在复制构造函数中，没有必要像重载赋值运算符函数中那样检查“自赋值”（相应地，这里指“自构造”）。因为 `int a = a;` 在 C++ 中是未定义的行为，就不应该出现，这种检查应该是编译器做的，而不是程序员应该做的。[3] [4]

4.2.3 重载赋值运算符时应该注意的问题

重载赋值运算符的一般步骤：

1. 检查“自赋值”，不检查自赋值可能会删除自身指针成员指向的内存。
2. 删除原成员内容。如果包含动态内存，需要使用 `delete`。

[3] https://stackoverflow.com/questions/2529111/stdstring-xx

[4] https://stackoverflow.com/questions/5517698/check-for-self-assignment-in-copy-constructor

3. 复制成员，这里和复制构造函数里的操作一致。如果涉及到动态内存，可以用 `new` 进行深拷贝。

4. 返回 `*this`。根据赋值运算符原本的特性，返回值一般设置为引用。

下面再详细地讨论一下返回值的问题。

返回值可以是 `void` 吗？不行，考虑 `a = b = c;` 如果返回值是 `void`，`a` 就不不能被正确赋值，所以必须返回 `*this`，以实现链式的表达式。

返回值为什么是 `MyString` 类型的引用呢？因为运算符重载时，我们希望尽量保留运算符原本的特性。例如 `(a = b) = c;` 这个语句，等价于 `(a.operator=(b)).operator=(c);` 按照赋值运算符的特性是应该改变 `a` 的值的，如果不返回该类型的引用，`a` 的值就不会改变。

返回值可以是 `MyString` 类型吗？可以，但是不好，原因上面已经提到，我们希望保留运算符原本的特性。我在自己尝试这种实现时还遇到了其他的问题，和复制构造函数有关。重载赋值运算符是通过成员函数实现的，而我们返回的又是 `*this`，即返回值是对象本身。前面学习复制构造函数时已经知道，一个类的对象作为函数的返回值，该函数返回时会调用该类的复制构造函数，产生一个临时的对象。例如执行 `s1 = s2;` 语句结束后，如果没有自己实现复制构造函数，就会调用默认的复制构造函数，通过浅拷贝产生 `s1` 的一个临时拷贝对象。由于并没有哪个变量去承接这个返回值，所以这个临时拷贝对象会立即调用析构函数并消亡。这时需要注意，在上面的例子中，由于没有自己实现深拷贝的复制构造函数，临时拷贝对象中的指针成员和 `s1` 中的指针成员就会指向同一块动态内存，临时拷贝对象消亡时，析构函数会把它们共同指向的这块动态内存删除。当 `s1` 对象也消亡时，会再次调用析构函数，可它指向的那块动态内存已经被删除了，这时就会报错。将复制构造函数设计为深拷贝就可以避免这个问题，但是每次返回都会进行一次对象的拷贝和释放，返回引用会更高效。

4.3 运算符重载为友元

运算符可以被重载为类的成员函数和普通函数，通常我们会将运算符重载为类的成员函数。有些时候，成员函数不能满足使用要求，普通函数又不能访问类的私有成员，这时候可以将运算符重载为友元函数。

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r, double i) : real(r), imag(i) {}
    Complex operator+(double r);
};

Complex Complex::operator+(double r) {
    return Complex(real + r, imag);
}
```

例如上面的代码定义了一个 `Complex` 类，并对 `+` 运算符进行了重载。经过上述重载后，可以对属于 `Complex` 类的对象 `c` 实现 `c = c + 5;` 的操作，相当于 `c = c.operator+(5);` 但是 `c = 5 + c;` 就会编译出错。为了使 `c = 5 + c;` 能成立，需要将 `+` 运算符重载为普通函数。

```
Complex operator+(double r, const Complex &c) {
    return Complex(c.real + r, c.imag);
}
```

但是 c.real 和 c.imag 都是要访问对象 c 的私有成员，普通函数是没有访问权限的。所以要把 + 运算符重载为友元函数。

4.4 流插入运算符和流提取运算符的重载

为什么 cout << 5 << "this"; 能够成立?

cout 是在 iostream 中定义的 ostream 类的对象。<< 能用在 cout 上是因为，在 iostream 里对 << 进行了重载。

怎么样才能使 cout << 5; 和 cout << "this" 都能成立? cout << 5; 即 cout.operator<<(5); cout << "this"; 即 cout.operator<<("this"); 所以该重载函数的返回值应该还是 cout。

```
ostream &ostream::operator<<(int n) {
    ...
    return *this;
}
```

假定下面程序输出为 5hello，应该怎么重载 << 运算符?

```
class CStudent {
public:
    int nAge;
};

int main(int argc, char const *argv[]) {
    CStudent s;
    s.nAge = 5;
    cout << s << "hello";    // 5hello
    return 0;
}
```

由于 cout 是 ostream 类的对象，而 ostream 是 iostream 中定义好的，所以我们不能将 << 重载为类的成员函数，只能将其重载为全局函数:

```
ostream &operator<<(ostream &o, const CStudent &s) {
    o << s.nAge;
    return o;
}
```

假定 c 是 Complex 复数类的对象，现在希望写 cout << c; 就能以 a+bi 的形式输出 c 的值，写 cin >> c; 就能从键盘接受 a+bi 形式的输入，并且使得 c.real = a, c.imag = b。

```
class Complex {
private:
    double real;
    double imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {};
    friend ostream &operator<<(ostream &os, const Complex &c);
    friend istream &operator>>(istream &is, Complex &c);
};

ostream &operator<<(ostream &os, const Complex &c) {
    os << c.real << "+" << c.imag << "i";
    return os;
}

istream &operator>>(istream &is, Complex &c) {
    string s;
    is >> s;
    int pos = s.find("+", 0);
    string sTmp = s.substr(0, pos); // get real part
    c.real = atof(sTmp.c_str());    // convert const char * to float
    sTmp = s.substr(pos+1, s.length()-pos-2);   // get imaginary part
    c.imag = atof(sTmp.c_str());
    return is;
}
```

4.5 自加、自减运算符的重载

自加运算符 ++ 和自减运算符 -- 有前置和后置之分。
前置运算符作为一元（单目）运算符重载：

- 重载为成员函数

```
T operator++();
T operator--();
```

- 重载为全局函数：

```
T operator++(T);
T operator--(T);
```

++obj、obj.operator++()、operator++(obj) 会调用上述函数。

后置运算符是也是一元（单目）运算符。但是后置运算符和前置运算符都是单目的，名字又一样，所以为了在重载时区分后置运算符与前置运算符，会引入一个多余的参数，将后置运算符作为二元（双目）运算符重载：

- 重载为成员函数

```
T operator++(int);
T operator--(int);
```

- 重载为全局函数

```
T operator++(T, int);
T operator--(T, int);
```

obj++、obj.operator++(0)、operator++(obj, 0) 会调用上述函数。

```
class CDemo {
private:
    int n;
public:
    CDemo(int i = 0) : n(i) {}
    CDemo &operator++();     // prefix increment
    CDemo operator++(int);  // postfix increment
    operator int() {return n;}  // int cast overloading
    friend CDemo &operator--(CDemo &);      // prefix decrement
    friend CDemo operator--(CDemo &, int);  // postfix decrement
};

// prefix increment
CDemo &CDemo::operator++() {
    n++;
    return *this;
}

// postfix increment
CDemo CDemo::operator++(int k) {
    CDemo tmp(*this);
    n++;
    return tmp;
}

// prefix decrement
CDemo &operator--(CDemo &d) {
    d.n--;
    return d;
}

// postfix decrement
```

```
CDemo operator--(CDemo &d, int) {
    CDemo tmp(d);
    d.n--;
    return tmp;
}
```

其中 operator int() {return n;} 语句是 int 作为一个强制类型转换运算符被重载。例如 Demo s; (int)s;

类型强制转换运算符重载时，不能写返回值类型，实际上其返回值类型就是强制类型转换运算符代表的类型。

4.6 运算符重载的注意事项

- C++ 不允许定义新的运算符
- 重载后运算符的含义应该符合日常习惯
  - complex_a + complex_b
  - word_a > word_b
  - date_b = date_a + n
- 运算符重载不改变运算符的优先级
- 以下运算符不能被重载：.，.*，::，? :，sizeof
- 重载运算符 ()、[]、-> 或者 = 时，重载函数必须声明为类的成员函数

5 继承和派生

5.1 继承和派生的概念

继承：在定义一个新的类 B 时，如果该类与某个已有的类 A 相似（指的是 B 拥有 A 的全部特点）。那么就可以把 A 作为一个基类，而把 B 作为基类的一个派生类（也称子类）。

派生类是通过对基类进行修改（覆盖）和扩充得到的。在派生类中，可以扩充新的成员变量和成员函数。派生类一经定义后，可以独立使用，不依赖于基类。

派生类拥有基类的全部成员函数和成员变量，不论是 private、protected 还是 public。在派生类的各个成员函数中，不能访问基类中的 private 成员。

派生类的写法：

```
class 派生类名 : public 基类名 {
    ...
};
```

派生类对象的体积，等于基类对象的体积，再加上派生类对象自己的成员变量的体积。在派生类对象中，包含着基类对象，而且基类对象的存储位置位于派生类对象之前。

5.2 继承关系和复合关系

类与类之间有三种关系：没关系、继承关系、复合关系。
继承："是"关系。

- 基类 A，B 是基类 A 的派生类
- 在逻辑上要求："一个 B 对象也是一个 A 对象"

复合："有"关系。

- 类 C 中"有"成员变量 k，k 是类 D 的对象，则 C 和 D 是复合关系
- 一般逻辑上要求："D 对象是 C 对象的固有属性或组成部分"

5.2.1 继承关系的使用

例如写了一个 CMan 类代表男人，后来发现又需要一个 CWoman 类代表女人，CWoman 类和 CMan 类有共同之处，就让 CWoman 类从 CMan 类派生而来，是否合适？不合适。因为"一个女人也是一个男人"从逻辑上不成立。

正确的做法是概括男人和女人的共同特点，写一个 CHuman 类，代表"人"，然后 CHuman 和 CWoman 都从 CHuman 派生。

5.2.2 复合关系的使用

例如在几何形体程序中，需要写"点"类，也需要写"圆"类，两者的关系就是复合关系——每一个"圆"对象里都包含一个"点"对象，这个"点"对象就是圆心。

```
class CPoint {
    double x, y;
    friend class CCircle;   // 便于 CCircle 类操作其圆心
};

class CCircle {
    double r;
    CPoint center;
};
```

如果要写一个小区养狗管理程序，需要写一个"业主"类，还需要写一个"狗"类。而狗是有"主人"的，主人即是业主。（假定狗只有一个主人，但一个业主最多可以有 10 条狗）

示例代码 1：

```
class CDog;
class CMaster {
    CDog dogs[10];
};
class CDog {
```

```
    CMaster m;
}
```

示例代码 1 是“人中有狗，狗中有人”，循环定义，无法得知应该给两个类的对象分配多大的空间，所以无法通过编译。

示例代码 2：

```
class CDog;
class CMaster {
    CDog *dogs[10];
};
class CDog {
    CMaster m;
};
```

示例代码 2 是“狗中有人”。即为“狗”类设一个“业主”类的成员对象，为“业主”类设一个“狗”类的对象指针数组。但是对于那些拥有相同主人的狗类对象来说，不容易使它们所包含的共同主人对象的信息保持一致。

示例代码 3：

```
class Master;
class CDog {
    CMaster *pm;
};
class CMaster {
    CDog dogs[10];
};
```

示例代码 3 是一种“凑合”的写法，为“狗”类设了一个“业主”类的对象指针，为“业主”类设了一个“狗”类的对象数组。但是从逻辑上讲，如果是复合关系，我们会要求一个类的成员对象时这个类的组成部分或固有属性，“狗”类对象不能说是“业主”组成部分或固有属性，所以从逻辑上来说这样设计也不太合适。

示例代码 4（正确写法）：

```
class CMaster;
class CDog {
    CMaster *pm;
};
class CMaster {
    CDog *dogs[10];
};
```

示例代码 4 是正确的写法，为“狗”类设一个“业主”类的对象指针，为“业主”类设一个“狗”类的对象指针数组。

5.3 基类和派生类有同名成员的情况

```
class base {
    int j;
public:
    int i;
    void func();
};

class derived : public base {
public:
    int i;
    void access();
    void func();
};

void derived::access() {
    j = 5;          // error, j is the private member of base class
    i = 5;          // OK, the i of derived class
    base::i = 5;    // OK, the i of base class
    func();         // OK, the func() of derived class
    base::func();   // OK, the func() of base class
}

int main() {
    derived obj;
    obj.i = 1;
    obj.base::i = 1;
}
```

derived 类的对象 obj 所占用的存储空间包括 Base::j、Base::i 和 i，即派生类中包含了基类的成员变量。

一般来说，基类和派生类不定义同名成员变量，但是有时候可以有同名成员函数。

5.4 访问范围说明符 protected

访问范围说明符有三种：public、private、protected，共同将一个类的成员分为三种，公有成员、私有成员和保护成员。

一个类的公有成员可以被任意访问。

一个类的私有成员只能由该类的成员函数或者友元函数访问，不允许在类外访问。派生类是不能直接访问基类的私有成员的，如果希望派生类能够访问它的成员，就把该成员声明为 public 或 protected。

保护成员允许其所在类、友元函数和派生类的成员函数访问。

基类的 private 成员，可以被下列函数访问：

- 基类的成员函数
- 基类的友元函数

基类的 `public` 成员，可以被下列函数访问：

- 基类的成员函数
- 基类的友元函数
- 派生类的成员函数
- 派生类的友元函数
- 其他的函数

基类的 `protected` 成员，可以被下列函数访问：

- 基类的成员函数
- 基类的友元函数
- 派生类的成员函数可以访问当前对象的基类的保护成员

保护成员可以直接被派生类访问到。这就意味着，如果改动了属于保护成员的某些属性，就需要修改基类和所有的派生类。因此，通常在派生自己写的基类时保护成员比较有用，而且保护成员的数目还不能太多。

5.5 派生类的构造函数

派生类对象包含基类对象，执行派生类构造函数之前，先执行基类的构造函数。

```
构造函数名(形参表) : 基类名(基类构造函数实参表)
{
    ...
}
```

在创建派生类的对象时：

- 需要调用基类的构造函数，初始化派生类对象中从基类继承的成员
- 在执行一个派生类的构造函数之前，总是先执行基类的构造函数

调用基类构造函数的两种方式：

- 显示方式：派生类的构造函数中 → 基类的构造函数提供参数

  ```
  derived::derived(arg_derived_list) : base(arg_base_list)
  ```

- 隐式方式：派生类的构造函数中省略基类构造函数时，派生类的构造函数会自动调用基类的默认构造函数。

派生类的析构函数被执行时，执行完派生类的析构函数后，自动调用基类的析构函数。
在创建派生类的对象时，执行派生类的构造函数之前：

- 调用基类的构造函数 → 初始化派生类对象中从基类继承的成员
- 调用成员对象类的构造函数 → 初始化派生类对象中的成员对象

执行完派生类的析构函数后：

- 调用成员对象类的析构函数
- 调用基类的析构函数

析构函数的调用顺序与构造函数的调用顺序相反。

5.6 public 继承的赋值兼容规则

```
class base {};
class derived : public base {};
base b;
derived d;
```

通过 public 方式派生时：

1. 派生类的对象可以赋值给基类对象

   ```
   b = d;
   ```

2. 派生类对象可以初始化基类引用

   ```
   base &br = d;
   ```

3. 派生类对象的地址可以赋值给基类指针

   ```
   base *pb = &d;
   ```

5.7 直接基类和间接基类

类 A 派生类 B，类 B 派生类 C，类 C 派生类 D，……

- 类 A 是类 B 的直接基类
- 类 B 是类 C 的直接基类，类 A 是类 C 的间接基类
- 类 C 是类 D 的直接基类，类 A、B 是类 D 的间接基类

在声明派生类时，只需要列出它的直接基类。派生类会沿着类的层次自动向上继承它的间接基类。
派生类的成员包括：

- 派生类自己定义的成员
- 直接基类中的所有成员
- 所有间接基类的的全部成员

6 虚函数和多态

多态是指同一名字的事物可以完成不同的功能，例如有几个相似而不完全相同的对象，有时人们要求在向它们发出同一个消息时，它们的反应各不相同，分别执行不同的操作。在 C++ 中，所谓多态性是指由继承而产生的相关的不同的类，其对象对同一消息会作出不同的响应。多态性是面向对象程序设计的一个重要特征，能有效增加程序的灵活性。

6.1 多态和虚函数的基本概念

6.1.1 虚函数

在类的定义中，前面有 `virtual` 关键字的成员函数就是虚函数。

```
class base {
    virtual int get();
};
int base::get() {}
```

`virtual` 关键字只用在类定义里的函数声明中，写函数体时不用。构造函数和静态成员函数不能是虚函数。

6.1.2 多态的表现形式一

派生类的指针可以赋值给基类指针。
通过基类指针调用基类和派生类中的同名虚函数时：

1. 若该指针指向一个基类的对象，那么被调用是基类的虚函数；
2. 若该指针指向一个派生类的对象，那么被调用的是派生类的虚函数。

这种机制就叫做“多态”。

```
class CBase {
public:
    virtual void someVirtualFunction() {}
};
class CDerived : public CBase {
public:
    virtual void someVirtualFunction() {}
};
int main() {
    CDerived oDerived;
    CBase *p = &oDerived;       // p points to oDerived (class CDerived)
    p->someVirtualFunction();   // someVirtualFunction in class CDerived
}
```

6.1.3 多态的表现形式二

派生类的对象可以赋值给基类引用。

通过基类引用调用基类和派生类中的同名虚函数时：

1. 若该引用引用的是一个基类的对象，那么被调用的是基类的虚函数；
2. 若该引用引用的是一个派生类的对象，那么被调用的是派生类的虚函数。

这种机制也叫做多态。

```
class CBase {
public:
    virtual void someVirtualFunction() {}
};
class CDerived : public CBase {
public:
    virtual void someVirtualFunction() {}
};
int main() {
    CDerived oDerived;
    CBase &r = oDerived;        // r refers to oDerived (class CDerived)
    r.someVirtualFunction();    // someVirtualFunction in class CDerived
}
```

在面向对象的程序设计中使用多态，能够增强程序的可扩充性，即程序需要修改或增加功能的时候，需要改动和增加的代码较少。

6.2 使用多态的游戏程序实例

游戏中有很多怪物，如战士、龙、凤凰、天使等，每种怪物都有一个类与之对应，每个怪物就是一个对象。怪物能够互相攻击，攻击敌人和被攻击时都有相应的动作，动作是通过对象的成员函数实现的。

在游戏版本升级时，要增加新的怪物——雷鸟。如何编程才能使升级时的代码改动和增加量较小?

基本思路是为每个怪物类编写 attack、fightBack 和 hurted 成员函数。其中 attack 函数表现攻击动作，攻击某个怪物，并调用被攻击怪物的 hurted 函数，以减少被攻击怪物的生命值，同时也调用被攻击怪物的 fightBack 成员函数，遭受被攻击怪物反击。hurted 函数减少自身生命值，并表现受伤动作。fightBack 成员函数表现反击动作，并调用被反击对象的 hurted 成员函数，使被反击对象受伤。

设置基类 CCreature，并且使 CDragon，CWolf 等其他类都从 CCreature 派生而来。

6.2.1 非多态的实现方式

有 n 种怪物，CDragon 类中就会有 n 个 attack 成员函数，以及 n 个 fightback 成员函数。对于其他类也如此。

如果游戏版本升级，增加了怪物雷鸟 CThunderBird，则程序改动较大。所有的类都需要增加两个成员函数：

- `void attack(CThunderBird *pThunderBird);`
- `void fightBack(CThunderBird *pThunderBird);`

在怪物种类多的时候，工作量将巨大无比！

6.2.2 多态的实现方式

基类 CCreature 只有一个 attack 成员函数；也只有一个 fightBack 成员函数；所有 CCreature 的派生类也是这样。如果游戏版本升级，增加了新的怪物雷鸟 CThunderBird，只需要编写新类 CThunderBird，不需要在已有的类里专门为新怪物增加成员函数：

- `void attack(CThunderBird *pThunderBird);`
- `void fightBack(CThunderBird *pThunderBird);`

而已有的类也可以原封不动。

```
class CCreature {
protected:
    int mLifeValue, mPower;
public:
    virtual void attack(CCreature *pCreature) {}
    virtual void hurted(int mPower) {}
    virtual void fightBack(CCreature *pCreature) {}
};

class CDragon : public CCreature {
public:
    virtual void attack(CCreature *pCreature);
    virtual void hurted(CCreature *pCreature);
    virtual void fightBack(CCreature *pCreature);
};

void CDragon::attack(CCreature *p) {
    // some attack codes
    p->hurted(mPower);
    p->fightBack(this);
}

void CDragon::hurted(int nPower) {
    // some hurted codes
    mLifeValue -= nPower;
}
```

```
void CDragon::fightBack(CCreature *p) {
    // some fight back codes
    p->hurted(mPower / 2);
}

int main(int argc, char const *argv[]) {
    CDragon dragon;
    CWolf wolf;
    CGhost ghost;
    CThunderBird thunderBird;
    dragon.attack(&wolf);           // CWolf::hurted ...
    dragon.attack(&ghost);          // CGhost::hurted ...
    dragon.attack(&thunderBird);    // CThunderBird::hurted ...
    return 0;
}
```

详见 “Week6/PolymorphismGameSample.cpp” 文件。

6.3 更多多态程序示例

6.3.1 几何形体处理程序

几何形体处理程序：输入若干个几何形体的参数，要求按面积排序输出。输出时要指明形状。
输入：第一行是几何形体数目 n （不超过 100）。下面有 n 行，每行以一个字母 c 开头。

- 若 c 是‘R’，则代表一个矩形，本行后面跟着两个整数，分别是矩形的宽和高
- 若 c 是‘C’，则代表一个圆，本行后面跟着一个整数代表其半径
- 若 c 是‘T’，则代表一个三角形，本行后面跟着三个整数，代表三条边的长度

输出：按面积从小到大依次输出每个几何形体的种类和面积。每行一个几何形体，输出格式为：形体名称: 面积

具体实现见 “Week6/GeometryWithPolymorphism.cpp” 文件。

用基类指针数组存放指向各种派生类对象的指针，然后遍历该数组，就能对各个派生类对象做各种操作，是很常用的做法。在上面的例子中，如果要添加一个新的几何形体五边形，则只需要从 `CShape` 类派生出 `CPentagon`，以及在 `main` 函数中的 `switch` 语句中增加一个 `case` 就可以了，其余部分都不用变。

6.4 多态的又一个例子

```
class Base {
public:
    void fun1() {
        // Call virtual function in a non-constructor and non-destructor
        // function will invoke polymorphism
```

```
        fun2();     // same as 'this->fun2();', this is a base pointer
    }
    virtual void fun2() {
        cout << "Base::fun2()" << endl;
    }
};

class Derived : public Base {
public:
    virtual void fun2() {
        cout << "Derived::fun2()" << endl;
    }
};

int main(int argc, char const *argv[]) {
    Derived d;
    // pBase is a pointer of base class, points to a object of derived class
    Base *pBase = &d;
    pBase->fun1();      // Derived::fun2()
    return 0;
}
```

在非构造函数、非析构函数的成员函数中调用虚函数，是多态。

在构造函数和析构函数中调用虚函数，不是多态。编译时即可确定调用的函数是自己的类或基类中定义的函数，不会等到运行时才决定调用自己的还是派生类的函数。

```
class Father {
public:
    virtual void hello() {
        cout << "Hello from Father." << endl;
    }
    virtual void bye() {
        cout << "Bye from Father." << endl;
    }
};

class Son : public Father {
public:
    Son() {hello();}    // hello is not polymorphism here
    ~Son() {bye();}     // derived bye() from base class Father
    // hello has the same virutal function name in base class, it's also virtual here
    void hello() {
```

```
        cout << "Hello from Son." << endl;
    }
};

class Grandson : public Son {
public:
    Grandson() {
        cout << "Constructing Grandson object" << endl;
    }
    ~Grandson() {
        cout << "Destructing Grandson object" << endl;
    }
    void hello() {  // virutal function
        cout << "Hello from Grandson" << endl;
    }
    void bye() {
        cout << "Bye from Grandson" << endl;
    }
};

int main(int argc, char const *argv[]) {
    Grandson grandson;
    Son *son;
    son = &grandson;
    son->hello();       // polymorphism
    // Hello from Son.
    // Constructing Grandson object
    // Hello from Grandson
    // Destructing Grandson object
    // Bye from Father.
    return 0;
}
```

6.5 多态的实现原理

“多态”的关键在于通过基类指针或引用调用一个虚函数时，编译时不确定到底调用的是基类还是派生类的函数，运行时才确定——这叫“动态联编”。“动态联编”是怎么实现的呢?

6.5.1 多态实现的关键——虚函数表

每一个有虚函数的类（或有虚函数的类的派生类）都有一个虚函数表，该类的任何对象都放着虚函数表的指针。虚函数表中列出了该类的虚函数地址。在创建含有虚函数的对象时，编译器会给其分配内

存来存放虚函数表的地址。

多态的函数调用语句被编译成一系列根据基类指针所指向的（或基类所引用的）对象中存放的虚函数表的地址，在虚函数表中查找虚函数的地址，并调用虚函数的指令。

6.5.2 实现多态的代价

多态能够有效提高程序的可扩充性，能够节省程序员的时间，但是使用多态也是有代价的。多态的程序在运行时会有额外的空间和时间的开销。因为每一个含有虚函数的类的对象都需要额外的空间来存储虚函数表的地址，在调用多态函数时，查找虚函数表的过程会有额外的时间上的开销。

在一个硬件便宜、人力贵的时代，让硬件多花点功夫，省下来人的时间，这是非常划算的。所以多态面向对象语言中一种非常好的特性。

6.6 虚析构函数

看看下面代码可能存在的问题：

```
class CSon {
public:
    ~CSon() {
        cout << "Bye CSon." << endl;
    }
};

class CGrandson : public CSon {
public:
    ~CGrandson() {
        cout << "Bye CGrandson" << endl;
    }
};

int main() {
    CSon *p = new CGrandson;
    delete p;
    // Bye CSon.
    return 0;
}
```

在上面的示例代码中，通过基类的指针 p 删除派生类的对象时，只会调用基类的析构函数，这就有可能会导致派生类对象中的动态内存泄漏。我们希望在删除一个派生类的对象时，能够先调用派生类的析构函数，然后再调用基类的析构函数。

解决办法是将基类的析构函数声明为 `virtual`。这样通过基类的指针删除派生类对象时，就会首先调用派生类的析构函数，然后调用基类的析构函数。

```
class CSon {
public:
```

```
    virtual ~CSon() {
        cout << "Bye CSon." << endl;
    }
};

class CGrandson : public CSon {
public:
    ~CGrandson() {
        cout << "Bye CGrandson" << endl;
    }
};

int main() {
    CSon *p = new CGrandson;
    delete p;
    // Bye CGrandson
    // Bye CSon.
    return 0;
}
```

由于派生类是继承自基类的，所以派生类的析构函数可以不用声明 `virtual`。
类中如果定义了虚函数，则最好将析构函数也定义成虚函数。
注意：不允许以虚函数作为构造函数。

6.7 纯虚函数和抽象类

纯虚函数：没有函数体的虚函数。

```
class A {
private:
    int a;
public:
    virtual void print() = 0;   // pure virtual function
    void fun() {cout << "fun";}
};
```

抽象类：包含纯虚函数的类。

- 只能作为基类来派生新类使用
- 不能创建抽象类的对象
- 抽象类的指针和引用 → 由抽象类派生出来类的对象

在抽象类中：

- 在成员函数内可以调用纯虚函数
- 在构造函数和析构函数内部不能调用纯虚函数

如果一个类从抽象类派生而来，它必须实现基类中所有的纯虚函数，它才能成为非抽象类。如果不实现所有的纯虚函数，该派生类就仍旧是抽象类，不能用它创建对象。

```
class A {
public:
    virtual void f() = 0;
    void g() {
        this->f();
    }
    // A() {f();}   // error: Pure virtual function called!
};

class B : public A {
public:
    void f() {
        cout << "B::f()" << endl;
    }
};

int main(int argc, char const *argv[]) {
    B b;
    b.g();
    // B::f()
    return 0;
}
```

7 文件操作和模板

7.1 文件操作

计算机中为了便于数据的管理和检索，引入了“文件”的概念。在程序中，要使用一个文件，先要打开，打开后才能读写，读写完要关闭。

数据的层次：位 → 字节 → 域/记录 → 文件

顺序文件：一个有限字符构成的顺序字符流。

C++ 标准库中共有 `ifstream`（继承自 `istream`）、`ofstream`（继承自 `ostream`）和 `fstream`（继承自 `iostream`）3 个类用于文件操作，统称为文件流类。

在 C++ 中使用或创建文件的基本流程：

1. 打开文件

- 通过指定文件名，建立文件和文件流对象的关联
- 指明文件的使用方式

2. 读/写文件
   - 利用读/写指针进行相应位置的操作
3. 关闭文件

7.1.1 建立顺序文件

```
#include <fstream>  // include the header
// Method 1 Open the file when initializing ofstream object
// "client.dat" is the file name, ios::out and ios::binary are file modes
ofstream outFile("client.dat", ios::out|ios::binary);
// Method 2 Open the file use the ofstream object's open() function
ofstream fout;
fout.open("test.out", ios::out|ios::binary);
// check the if the file has been opened
if (!fout) { cerr << "File open error!" << endl; }
```

其中文件名可以是绝对路径，也可以是相对路径。如果没有交代路径信息，就是在当前目录下查找文件。

7.1.2 文件的读写指针

对于输入文件，有一个读指针；对于输出文件，有一个写指针；对于输入输出文件，有一个读写指针。上述指针可以用于标识文件操作的当前位置，指针在哪里，读写操作就在哪里进行。

```
// write pointer
ofstream fout("a1.out", ios::app);  // append mode
long location = fout.tellp();       // get the file write pointer
// the location can be positive or negative
location = 10L;
// move file pointer to the 10th character
fout.seekp(location);
// the offset is relative to the the beginning of the file (default)
fout.seekp(location, ios::beg);
// the offset is relative to the current location of the file pointer
fout.seekp(location, ios::cur);
// the offset is relative to the end of the file
fout.seekp(location, ios::end);

// read pointer
ifstream fin("a1.in", ios::in);     // read mode
long location = fin.tellg();        // get the file read pointer
```

```
location = 10L;
fin.seekg(location);
fin.seekg(location, ios::beg);
fin.seekg(location, ios::cur);
fin.seekg(location, ios::end);
```

7.1.3 二进制文件的读写

```
int x = 10;
fout.seekp(20, ios::beg);
fout.write((const char *)(&x), sizeof(int));

fin.seekg(0, ios::begin);
fin.read((char *)(&x), sizeof(int));
```

在对二进制文件读写时，直接读写二进制数据，未必能用记事本等编辑器正确显示。
无论是读还是写文件，都需要显示地关闭文件，否则可能会导致文件内容出现问题。

7.2 函数模板

7.2.1 泛型程序设计的概念

泛型程序设计（Generic Programming）：算法实现时不指定具体要操作的数据类型。在进行泛型程序设计时，仅需要用算法实现一遍，就可以适用于多种数据结构。
泛型设计的优势在于可以减少重复代码的编写。
模板分为两类：函数模板和类模板。
泛型程序设计是一个非常重要的概念，可以和前面的抽象、封装、继承、多态相提并论。

7.2.2 函数模板的编写

为了交换两个 `int` 型变量的值，需要编写一个 `void swap(int &x, int &y)` 的函数。如果现在又想交换两个 `double` 型的变量的值，还需要编写 `void swap(double &x, double &y)` 函数。能否只写一个 `swap` 函数，就能交换各种类型的变量呢？答案是可以，通过函数模板来实现。

```
template<class 类型参数 1, class 类型参数2, ...>
返回值类型 函数模板名(形参表) {
    // 函数体
}
```

交换两个变量值的函数模板：

```
template<class T>
void swap(T &x, T &y) {
    T tmp = x;
    x = y;
```

```
    y = tmp;
}

int main() {
    int n = 1, m = 2;
    swap(n, m);
    double f = 1.2, g = 2.3;
    swap(f, g);
}
```

函数模板中可以有不止一个类型参数：

```
template<class T1, class T2>
T2 print(T1 arg1, T2, arg2) {
    cout << arg1 << " " << arg2 << endl;
    return arg2;
}
```

7.2.3 函数模板的重载

函数模板可以重载，只要它们的形参表不同即可。例如：

```
template <class T1, class T2>
void print(T1 arg1, T2 arg2) {
    cout << "template 1 is called." << endl;
    cout << arg1 << " " << arg2 << endl;
}

template <class T>
void print(T arg1, T arg2) {
    cout << "template 2 is called." << endl;
    cout << arg1 << " " << arg2 << endl;
}

int main(int argc, char const *argv[]) {
    int a = 1, b = 2;
    double c = 3.14;
    print(a, b);
    // template 2 is called.
    // 1 2
    print(a, c);
    // template 1 is called.
    // 1 3.14
```

```
    return 0;
}
```

7.2.4 C++ 函数重载时匹配的优先级

现在学了函数模板再做重载的时候，我们就会发现，现在一个程序中可能会同时出现同名的普通函数和同名的模板函数。C++ 编译器是如何匹配它们的呢?

C++ 编译器遵循以下优先顺序:

1. 先找参数完全匹配的普通函数（非模板函数）
2. 再找参数完全匹配的模板函数
3. 再找实参经过自动类型转换后能够匹配的普通函数
4. 上面的都没找到，报错

```
template <class T>
T myMax(T a, T b) {
    cout << "template 1 is called" << endl;
    return 0;
}

template <class T1, class T2>
T1 myMax(T1 a, T2 b) {
    cout << "template 2 is called" << endl;
    return 0;
}

double myMax(double a, double b) {
    cout << "normal myMax is called" << endl;
    return 0;
}

int main(int argc, char const *argv[]) {
    int i = 4, j = 5;
    myMax(1.2, 3.4);    // normal myMax is called
    myMax(i, j);        // template 1 is called
    myMax(1.2, 3);      // template 2 is called
    return 0;
}
```

7.2.5 赋值兼容原则引起函数模板中类型参数的二义性

```
template <class T>
T myFunction(T arg1, T arg2) {
```

```
    cout << arg1 << " " << arg2 << endl;
    return arg1;
}

int main() {
    myFunction(5, 7);       // OK, replace T with int
    myFunction(5.8, 8.4);   // OK, replace T with double

    // error: no matching function for call to 'myFunction'
    // candidate template ignored: deduced conflicting types for
    // parameter 'T' ('int' vs. 'double')
    myFunction(5, 8.4);
}
```

可以在函数模板中使用多个类型参数，可以避免二义性。

```
template <class T1, class T2>
T1 myFunction(T1 arg1, T2 arg2) {
    cout << arg1 << " " << arg2 << endl;
    return arg1;
}

int main() {
    myFunction(5, 7);       // OK, replace T1 and T2 with int
    myFunction(5.8, 8.4);   // OK, replace T1 and T2 with double
    myFunction(5, 8.4);     // OK, replace T1 with int, replace T2 with double
}
```

7.2.6 函数模板的特化

在我们使用函数模板时，编译器会根据传入函数模板的实参对其实例化，将模板参数类型替换为实参的类型。这就意味着，虽然由于实参的不同会导致不同的数据类型，但同一个函数模板的实现方式是统一的。

但是有些时候，我们希望针对某个特定数据类型，使模板采用不同的实现方式。这就是模板的特化（template specialization）。

函数模板特化的语法：

```
template <>
返回值类型 函数名<指定数据类型>(参数表) {
    ...
}
```

函数模板特化的代码中就不能出现类型参数 T 了，应该用指定的数据类型代替。

例如，我们有一个可以打印不同参数的函数模板：

```
template <class T>
void print(T a) {
    cout << a << endl;
}

int main(int argc, char const *argv[]) {
    int a = 1;
    double b = 2.0;
    char c = 'C';
    string d = "Dog";
    print(a);   // 1
    print(b);   // 2
    print(c);   // C
    print(d);   // Dog
    return 0;
}
```

现在我们想在打印 double 类型的数据时显示为科学计数法。这时就可以针对double 类型，对函数模板进行特化，这时可以在上面的代码中插入以下代码：

```
template <>
void print<double>(double a) {
    cout << scientific << a << endl;
}
```

再次运行时，print(b); 的结果就会由原来的 2 变成 2.000000e+00。

7.3 类模板

众多函数可以写成一个函数模板，若干个类也可以写成一个模板，这样可以使程序进一步地简化。

定义一批相似的类 → 定义类模板 → 生成不同的类

7.3.1 类模板的定义

首先通过一个问题来引入类模板。数组是一种常见的数据类型，其元素可以是整型、字符串或者我们自定义的类型。现考虑设计一个数组类，需要提供基本的如查看数组长度、访问、修改其中的某个元素等操作。对于这些数组类，除了元素的类型不同之外，其他的完全相同。

在定义类模板的时候需要给它一个/多个参数，这个/些参数表示不同的数据类型。在调用类模板时，指定参数，由编译系统根据参数提供的数据类型自动产生相应的模板类。

C++ 的类模板写法如下：

```
template <类型参数表>
class 类模板名 {
    // 成员函数和成员变量
};
```

如果在类模板外定义类模板的成员函数时：

```
template <类型参数表>
返回值类型 类模板名<类型参数名列表>::成员函数表(参数表) {
    ...
}
```

用类模板定义对象的写法如下：

```
类模板名<真实类型参数表> 对象名(构造函数实际参数表);
```

如果类模板有无参构造函数，那么也可以只写：`类模板名<真实类型参数表> 对象名;`

Pair 类模板：

```
template <class T1, class T2>
class Pair {
public:
    T1 key;
    T2 value;
    Pair(T1 k, T2 v) : key(k), value(v) {};
    bool operator<(const Pair<T1, T2> &p) const;
};

template <class T1, class T2>
bool Pair<T1, T2>::operator<(const Pair<T1, T2> &p) const {
    return key < p.key;
}

int main(int argc, char const *argv[]) {
    Pair<string, int> student("Tom", 19);
    cout << student.key << " " << student.value << endl;    // Tom 19
    return 0;
}
```

7.3.2 使用模板声明对象

编译器由类模板生成类的过程叫做类模板的实例化。编译器会自动用具体的数据类型来替换类模板中的类型参数，生成模板类的代码。

由类模板实例化得到的类叫模板类。为类型参数指定的数据类型不同，得到的模板类不同。

同一个类模板的两个模板类是不兼容的：

```
Pair<stirng, int> *p;
Pair<stirng, double> a;
p = &a;     // wrong
```

7.3.3 函数模板作为类模板成员

```
template <class T>
class A {
public:
    template <class T2>
    void func(T2 t) { cout << t << endl; }  // member function template
};

int main(int argc, char const *argv[]) {
    A<int> a;
    a.func('K');    // instantiate the member function template
    return 0;
}
```

7.3.4 类模板的特化

类模板的特化和函数模板的特化非常相似，同样是为了模板在针对某些特定数据类型的时候采用不同的实现方式。

例如下面的模板类，它能打印 MyClass 构造函数的参数：

```
template <class T>
class MyClass {
public:
    MyClass(T x) {
        cout << x << " - not a char" << endl;
    }
};
```

现在我们希望该类模板在类型参数 T 为 char 时能够做一些不同的操作，这时可以针对 char 类型对该类模板进行特化，插入以下代码：

```
template <>
class MyClass<char> {
public:
    MyClass(char x) {
        cout << x << " is a char!" << endl;
    }
};

int main(int argc, char const *argv[]) {
    MyClass<int> obj1(42);          // 42 - not a char
    MyClass<double> obj2(5.47);     // 5.47 - not a char
    MyClass<char> obj3('s');        // s is a char!
```

```
    return 0;
}
```

需要注意，特化模板和通用模板之间不存在“继承”关系，所以特化模板的类的成员仍需自行定义。

7.3.5 类模板与非类型参数

类模板的参数声明中可以包括非类型参数：`template <class T, int elementsNumber>`，这也叫作模板的部分特化。

- 非类型参数：用来说明类模板中的属性
- 类型参数：用来说明类模板中的属性类型，成员从操作的参数类型和返回值类型

```
template <class T, int size>
class CArray {
public:
    T array[size];
    void print() {
        for (int i = 0; i < size; ++i) {
            cout << array[i] << endl;
        }
    }
};
```

非类型参数的参数类型是某个固定的类型，在实例化时要指定为具体的值。

注意：非类型参数不同的了两个模板类也是不同的类。例如 `CArray<int, 40>` 和 `CArray<int, 50>` 完全是两个类，这两个类的对象之间不能互相赋值。

7.3.6 类模板的继承与派生

类模板继承的四种形式：

1. 类模板派生出类模板

```
template <class T1, class T2>
class A1 {
protected:  // private members can't be directly accessed by derived class
    T1 v1;
    T2 v2;
public:
    A1(T1 val1, T2 val2) : v1(val1), v2(val2) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        // execute 'a.out | c++fill' to get right display of 'name'
```

```
        cout << "T1 v1: " << typeid(v1).name() << " " << v1 << endl;
        cout << "T2 v2: " << typeid(v2).name() << " " << v2 << endl;
    }
};

template <class T1, class T2>
class A2 : public A1<T2, T1> {    // note the sequence of T1 and T2
protected:
    T1 v3;
    T2 v4;
public:
    A2(T2 val1, T1 val2, T1 val3, T2 val4)
        : A1<T2, T1>(val1, val2), v3(val3), v4(val4) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        // cannot access directly. see:
        // https://isocpp.org/wiki/faq/templates#nondependent-name-lookup-members
        // https://stackoverflow.com/questions/24158209/
        cout << "T2 v1: " << typeid(this->v1).name() << " " << this->v1 << endl;
        cout << "T1 v2: " << typeid(this->v2).name() << " " << this->v2 << endl;
        cout << "T1 v3: " << typeid(v3).name() << " " << v3 << endl;
        cout << "T2 v4: " << typeid(v4).name() << " " << v4 << endl;
    }
};

int main() {
    A1<int, double> a1(1, 2.0);         // int v1, double v2
    // Class: A1<int, double>
    a1.printMemberVarsTypes();
    // T1 v1: int 1
    // T2 v2: double 2
    A2<int, double> a2(3.0, 4, 5, 6.0); // double v1, int v2, int v3, double v4
    // Class: A1<double, int>
    // Class: A2<int, double>
    a2.printMemberVarsTypes();
    // T2 v1: double 3
    // T1 v2: int 4
    // T1 v3: int 5
    // T2 v4: double 6
    return 0;
```

```
}
```

2. 模板类派生出类模板。模板类是类模板中类型/非类型参数实例化后的类，类型参数实例化后变成某一种具体的数据类型，非类型参数实例化后会变成某一个具体的值或对象，例如 A<int, double> 就是一个模板类。

```
template <class T1, class T2>
class B1 {
protected:
    T1 v1;
    T2 v2;
public:
    B1(T1 val1, T2 val2) : v1(val1), v2(val2) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        cout << "T1 v1: " << typeid(v1).name() << " " << v1 << endl;
        cout << "T2 v2: " << typeid(v2).name() << " " << v2 << endl;
    }
};

template <class T>
class B2 : public B1<int, double> {
private:
    T v3;
public:
    B2(int val1, double val2, T val3) : B1<int, double>(val1, val2), v3(val3) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        // no need to use this pointer to access v1, v2 here
        cout << "int v1: " << typeid(v1).name() << " " << v1 << endl;
        cout << "double v2: " << typeid(v2).name() << " " << v2 << endl;
        cout << "T3 v3: " << typeid(v3).name() << " " << v3 << endl;
    }
};

int main() {
    B1<double, char *> b1(1.0, "hello");
    // Class: B1<double, char*>
    b1.printMemberVarsTypes();
    // T1 v1: double 1
```

```
    // T2 v2: char* hello
    B2<char> b2(3, 4.0, 'E');
    // Class: B1<int, double>
    // Class: B2<char>
    b2.printMemberVarsTypes();
    // int v1: int 3
    // double v2: double 4
    // T3 v3: char E
    return 0;
}
```

3. 普通类派生出模板类

```
class C1 {
protected:
    int v1;
public:
    C1(int val1) : v1(val1) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        cout << "int v1: " << typeid(v1).name() << " " << v1 << endl;
    }
};

template <class T>
class C2 : public C1 {
private:
    T v2;
public:
    C2(int val1, T val2) : C1(val1), v2(val2) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        cout << "int v1: " << typeid(v1).name() << " " << v1 << endl;
        cout << "T v2: " << typeid(v2).name() << " " << v2 << endl;
    }
};

int main() {
    C1 c1(1);
    // Class: C1
```

```
    c1.printMemberVarsTypes();
    // int v1: int 1
    C2<char> c2(2, 'C');
    // Class: C1
    // Class: C2<char>
    c2.printMemberVarsTypes();
    // int v1: int 2
    // T v2: char C
}
```

4. 模板类派生出普通类

```
template <class T>
class D1 {
protected:
    T v1;
    int v2;
public:
    D1(T val1, int val2) : v1(val1), v2(val2) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        cout << "T v1: " << typeid(v1).name() << " " << v1 << endl;
        cout << "int v2: " << typeid(v2).name() << " " << v2 << endl;
    }
};

class D2 : public D1<int> {
private:
    double v3;
public:
    D2(int val1, int val2, double val3) : D1(val1, val2), v3(val3) {
        cout << "Class: " << typeid(*this).name() << endl;
    }
    void printMemberVarsTypes() {
        cout << "int v1: " << typeid(v1).name() << " " << v1 << endl;
        cout << "int v2: " << typeid(v2).name() << " " << v2 << endl;
        cout << "double v3: " << typeid(v3).name() << " " << v3 << endl;
    }
};

int main() {
```

```
        D1<char> d1('A', 2);
        // Class: D1<int>
        d1.printMemberVarsTypes();
        // T v1: char A
        // int v2: int 2
        D2 d2(3, 4, 5);
        // Class: D1<int>
        // Class: D2
        d2.printMemberVarsTypes();
        // int v1: int 3
        // int v2: int 4
        // double v3: double 5
    }
```

7.4 string 类

7.4.1 string 类的初始化

string 类是一个模板类，它的定义如下：

```
typedef basic_string<char> string;
```

string 对象的初始化：

- `string s1("Hello");`
- `string s2(8, 'x');`
- `string month = "March";`

string 类不提供以字符和整数为参数的构造函数。
错误的初始化方法：

- `string error1 = 'c';`
- `string error2('u');`
- `string error3 = 22;`
- `string error4(8);`

但是可以将字符赋值给 string 对象：`string s; s = 'n';`
如果构造的 string 过长而无法表达，会抛出 length error 的异常。
string 对象的长度用成员函数 length() 读取。

```
string s("hello");
cout << s.length() << endl;
```

string 支持流读取运算符 <<。

```
string strObj;
cin >> strObj;
```

string 支持 getline() 函数。

```
string s;
getline(cin, s);
```

7.4.2 string 的赋值和链接

- 用“=”赋值

```
string s1("cat"), s2;
s2 = s1;
```

- 用 assign 成员函数赋值

```
string s1("cat"), s3;
s3.assign(s1);
```

- 用 assign 成员函数部分赋值

```
string s1("catpig"), s3;
s3.assign(s1, 1, 3);    // start from the 1st char of s1, copy 3 chars to s3
```

- 单个字符赋值

```
s2[5] = s1[3] = 'a';
```

- 逐个访问 string 对象中的字符

```
string s1("Hello");
for (int i = 0; i < s1.length(); i++)
    cout << s1.at(i) << endl;
```

下标运算符 [] 和成员函数 at 的区别：下标运算符 [] 不做范围检查；成员函数 at 做范围检查，如果超出范围会抛出 out of range 的异常。

- 用 + 运算符连接字符串

```
string s1("good"), s2("morning!");
s1 += s2;
cout << s1 << endl;
```

- 用成员函数 append 连接字符串

```
string s1("good"), s2("morning!");
s1.append(s2);          // goodmorning!
cout << s1 << endl;
// start from the 3rd char of s1, copy s1.size() chars to the end of s2
```

```
// if there are not so many chars in s1, end copying at the end of s1.
s2.append(s1, 3, s1.size());
cout << s2 << endl;     // morning!dmorning!
```

7.4.3 string 的比较

- 用关系运算符 ==、>、>=、<、<=、!= 比较 string 的大小。比较表达式的返回值都是 bool 类型，成立返回 true，否则返回 false。
- 用成员函数 compare 比较 string 的大小。

```
string s1("hello"), s2("hello"), s3("hell");
int f1 = s1.compare(s2);
int f2 = s1.compare(s3);
int f3 = s3.compare(s1);
// int compare (size_t pos, size_t len, const string& str,
//       size_t subpos, size_t sublen) const;
int f4 = s1.compare(1, 2, s3, 0, 3);
// int compare (size_t pos, size_t len, const char* s) const;
int f5 = s1.compare(0, s1.size(), s3);
cout << f1 << " " << f2 << " " << f3 << " ";
cout << f4 << " " << f5 << endl;
// 0 1 -1 -1 1 (on vc++)
// 0 1 -1 -3 1 (on gcc/clang)
```

7.4.4 子串

- 成员函数 substr() 可以用于截取一个 string 对象的子串。

```
string s1("hello world"), s2;
// string substr (size_t pos = 0, size_t len = npos) const;
s2 = s1.substr(4, 5);
cout << s2 << endl;
```

7.4.5 交换 string

- 成员函数 swap() 可以用于交换两个 string 的值。

```
string s1("hello world"), s2("really");
s1.swap(s2);
cout << s1 << endl;     // really
cout << s2 << endl;     // hello world
```

7.4.6 string 的特性

- 成员函数 capacity()，返回无需增加内存即可存放的字符数

- 成员函数 maximum_size()，返回 string 对象可存放的最大字符数
- 成员函数 length() 和 size() 相同，返回字符串的大小/长度
- 成员函数 empty()，返回 string 对象是否为空
- 成员函数 resize()，改变 string 对象的长度

7.4.7 寻找 string 中的字符

- 成员函数 find()

```
string s1("hello world");
// 在 s1 中从前向后查找“lo”第一次出现的地方。如果找到，返回“lo”开始的位置，即‘l’
// 所在的位置下标；如果找不到，返回 \mintinline{C++}{string::npos}
cout << s1.find("lo") << endl;  // 3
```

- 成员函数 rfind()

```
string s1("hello world");
// 在 s1 中从后向前查找“lo”第一次出现的地方。如果找到，返回“lo”开始的位置，即‘l’
// 所在位置的下标；如果找不到，返回 \mintinline{C++}{string::npos}
cout << s1.rfind("lo") << endl; // 3
```

- 成员函数 find_first_of()

```
string s1("hello world");
// 在 s1 中从前向后查找“abcd”中任何一个字符第一次出现的地方。如果找到，返回找到
// 字母的位置；如果找不到，返回 \mintinline{C++}{string::npos}
cout << s1.find_first_of("abcd") << endl;       // 10
```

- 成员函数 find_last_of()

```
string s1("hello world");
// 在 s1 中从后向前查找“abcd”中任何一个字符最后一次出现的地方。如果找到，返回找到
// 字母的位置；如果找不到，返回 \mintinline{C++}{string::npos}
cout << s1.find_last_of("abcd") << endl;        // 10
```

- 成员函数 find_first_not_of()

```
string s1("hello world");
// 在 s1 中从前向后查找不在“abcd”中的字符第一次出现的地方。如果找到，返回找到
// 字母的位置；如果找不到，返回 \mintinline{C++}{string::npos}
cout << s1.find_first_not_of("abcd") << endl;   // 0
```

- 成员函数 find_last_not_of()

```
string s1("hello world");
// 在 s1 中从后向前查找不在“abcd”中的字符最后一次出现的地方。如果找到，返回找到
// 字母的位置；如果找不到，返回 \mintinline{C++}{string::npos}
cout << s1.find_last_not_of("abcd") << endl;    // 9
```

7.4.8 替换 string 中的字符

- 成员函数 erase()

```
string s1("hello worlld");
s1.erase(5);    // 清除 s1 下标 5 之后的所有内容
cout << s1 << endl;             // hello
cout << s1.length() << endl;    // 5
cout << s1.size() << endl;      // 5
```

- 成员函数 find()

```
s1 = "hello worlld";
// 分别从下标 1、2、3 的位置寻找“ll”
cout << s1.find("ll", 1) << endl;   // 2
cout << s1.find("ll", 2) << endl;   // 2
cout << s1.find("ll", 3) << endl;   // 9
```

- 成员函数 replace()

```
s1 = "hello worlld";
// 将 s1 中下标 2 开始的 3 个字符替换成“haha”
s1.replace(2, 3, "haha");
cout << s1 << endl;                 // hehaha worlld
s1 = "hello worlld";
// 将 s1 中下标 2 开始的 3 个字符替换成“haha”中下标 1 开始的 2 个字符
s1.replace(2, 3, "haha", 1, 2);
cout << s1 << endl;                 // heah worlld
```

7.4.9 在 string 中插入字符

- 成员函数 insert()

```
s1 = "hello world";
string s2("show insert");
// 将 s2 插入 s1 下标 5 的位置
s1.insert(5, s2);
cout << s1 << endl;         // helloshow insert world
// 将 s2 中下标 5 开始的 3 个字符插入 s1 下标 2 的位置
s1.insert(2, s2, 5, 3);
cout << s1 << endl;         // heinslloshow insert world
```

7.4.10 将 string 转换成 C 语言式 char * 字符串

- 成员函数 c_str()

```
s1 = "hello world";
// s1.c_str() 返回传统的 const char * 类型的字符串，且该字符串以'\0'结尾
printf("%s\n", s1.c_str()); // hello world
```

- 成员函数 data()

```
s1 = "hello world";
// s1.data() 返回一个 char * 类型的字符串，对 s1 的修改可能会使 p1 出错
const char *p1 = s1.data();
for (int i = 0; i < s1.length(); i++) {
    printf("%c", *(p1 + i));
}                           // hello world
cout << endl;
```

- 成员函数 copy()

```
s1 = "hello world";
int len = s1.length();
char *p2 = new char[len+1];
// s1.copy(p2, 5, 0) 从 s1 的下标 0 的字符开始，制作一个最长 5 个字符的字符副本
// 并将其赋值给 p2，返回值表明实际复制字符串的长度
s1.copy(p2, 5, 0);
p2[5] = 0;
cout << p2 << endl;         // hello
```

7.5 输入输出

与输入输出流操作相关的类：ios、istream、ostream、iostream、ifstream、ofstream、fstream。

7.5.1 标准流对象

- 输入流对象：
  - cin 与标准输入设备相连。cin 对应于标准输入流，用于从键盘读取数据，也可以被重定向为从文件中读取数据。
- 输出流对象：
  - cout 与标准输出设备相连。cout 对应于标准输出流，用于向屏幕输出数据，也可以被重定向为向文件写入数据。
  - cerr 与标准错误输出设备相连。cerr 对应于标准错误输出流，用于向屏幕输出错误信息。
  - clog 与标准错误输出设备。clog 对应于标准错误输出流，用于向屏幕输出错误信息。
    cerr 和 clog 的区别在于 cerr 不使用缓存区，直接向显示器输出信息；而输出到 clog 中的信息先会被存放在缓冲区，缓冲区满或者刷新时才输出到屏幕。

7.5.2 输入、输出重定向

```
double f;
int n;
freopen("ind.txt", "r", stdin);       // redirect stdin
cin >> f >> n;
freopen("out.txt", "w", stdout);     // redirect stdout
if (n == 0) {
    cerr << "error" << endl;         // won't print to "out.txt"
}
else {
    cout << f / n << endl;
}

fclose(stdin);
fclose(stdout);
```

7.5.3 判断输入流结束

可以利用如下方法判断输入流结束：

```
int x;
while (cin >> x) {
    ...
}
```

- 如果是从文件输入，比如前面有 `freopen("some.txt", "r", stdin);` 那么，读到文件尾部，输入流就算结束，跳出循环
- 如果从键盘输入，则在单独一行输入 `Ctrl + Z` 代表输入流结束，跳出循环

前面已经接触到，`cin` 返回的是 `cin` 的引用，但这里它会强制转换成 `bool` 类型。
也可以利用 `istream` 类的成员函数来判断输入流是否结束。

- `istream &getline(char *buf, int bufSize);`

  从输入流中读取 `bufSize - 1` 个字符到缓存区 `buf`，或读到碰到 `'\n'` 为止（哪个先到算哪个）

- `istream &getline(char *buf, int bufSize, char delim);`

  从输入流中读取 `bufSize - 1` 个字符到缓存区 `buf`，或读到碰到 `delim` 字符为止（哪个先到算哪个）

两个函数都会自动在 `buf` 中读入数据的结尾添加 `'\0'`。`'\n'` 或 `delim` 都不会被读入 `buf`，但会被从输入流中取走。如果输入流中 `'\n'` 或 `delim` 之前的字符个数达到或超过了 `bufSize` 个，就导致读入出错。其结果是：虽然本次读入已经完成，但是之后的读入就都会失败了。

可以用 `if (!cin.getline(...))` 判断输入是否结束。

示例代码：

```
int x;
char buf[100];
cin >> x;
cin.getline(buf, 90);
cout << buf << endl;
```

上面的示例代码在运行时会出现一个比较奇怪的现象：如果输入“12 abcd”，会输出“abcd”；但是如果希望先输入“12”后回车，再输入“abcd”时，就会发现输入回车程序立马就会终止。这是因为 `getline` 在读到输入流中的 `'\n'` 时会立刻返回。

`bool eof();` 判断输入流是否结束。

`int peek();` 返回下一个字符，但不会从流中去掉。

`istream &putback(char c);` 将字符 `ch` 放回输入流。

`istream &ignore(int nCount = 1, int delim = EOF);` 从流中删掉最多 `nCount` 个字符，遇到 `EOF` 时结束。

8 标准模板库 STL

本模块将介绍 C++ 的标准模板库（Standard Template Library, STL），这是泛型程序设计最成功的应用实例。STL 是一些常用数据结构（如链表、可变长数据、排序二叉树）和算法（如排序、查找）的模板的集合。有了 STL，我们就不必编写大多的、常用的数据结构和算法，并且可获得非常高的性能。

8.1 STL 概述

C++ 语言的核心优势之一就是便于软件的重用。

C++ 中有两个方面体现软件的重用：

1. 面向对象的思想：继承和多态，标准类库
2. 泛型程序设计（generic programming）的思想：模板机制，以及标准模板库 STL

STL 中的一些基本概念：

容器 可容纳各种数据类型的通用数据结构，是类模板

迭代器 可用于依次存取容器中的元素，类似于指针

算法 用来操作容器中的元素的函数模板

- `sort()` 用来对一个 `vector` 中的数据进行排序
- `find()` 用来搜索一个 `list` 中的对象

算法本身与他们操作的数据类型无关，因此它们可以在从简单数据到高度复杂容器的任何数据结构上使用。

示例代码：

```
int array[100];
sort(array, array+70);  // 将前 70 个元素排序
```

其中，数组 array[100] 就是容器，而 `int *` 类型的指针变量就可以作为迭代器，`sort` 算法就可以作用于该容器上，对其进行排序。

8.2 容器概述

容器可以用于存放各种类型的数据（基本类型的变量、对象等）的数据结构，都是类模板，分为三种：

1. 顺序容器：`vector`，`deque`，`list`
2. 关联容器：`set`，`multiset`，`map`，`multimap`
3. 容器适配器：`stack`，`queue`，`priority_queue`

对象被插入容器中，被插入的对象是一个复制品。许多算法，比如排序、查找等，要求对容器中的元素进行比较，有的容器本身就是排序的，所以，放入容器的对象所属的类，往往还应该重载 `==` 和 `<` 运算符。

8.2.1 顺序容器简介

顺序容器有 `vector`、`deque`、`list` 三种。顺序容器不是排序的，元素的插入位置同元素的值无关。

- `vector` 头文件 `<vector>`
  动态数组。元素在内存中连续存放。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能（大部分情况是常数时间）。在头端和中间插入元素时间复杂度为 $O(n)$。
- `deque` 头文件 `<deque>`
  双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成（但次于 `vector`）。在两端增删元素具有较佳的性能（大部分情况是常数时间）。
- `list` 头文件 `<list>`
  双向链表。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成（前提是已找到增删元素的位置）。不支持随机存取。

8.2.2 关联容器简介

关联容器是排序的，插入任何元素，都按相应的排序规则来确定其位置。
关联容器在查找时具有非常好的性能，通常以平衡二叉树的方式实现，插入和检索的时间都是 $O(log(N))$。

- `set/multiset` 头文件 `<set>`
  集合。`set` 中不允许相同元素，`multiset` 中允许存在相同的元素。
- `map/multimap` 头文件 `<map>`
  `map` 与 `set` 不同在于 `map` 中存放的元素有且仅有两个成员变量，一个名为 `first`，另一个名为 `second`，`map` 根据 `first` 值对元素进行从小到大排序，并可快速地根据 `first` 来检索元素。`map` 同 `multimap` 的不同在于是否允许相同 `first` 值的元素。

8.2.3 容器适配器简介

- stack 头文件 <stack>
  栈。stack 是项的有限序列，并满足序列中被删除、检索和修改的项只能是最近插入序列的项（栈顶的项）。后进先出。
- queue 头文件 <queue>
  队列。插入只可以在尾部进行，删除、检索和修改只允许从头部进行。先进先出。
- priority_queue 头文件 <queue>
  优先级队列。最高优先级元素总是第一个出列。

8.2.4 顺序容器和关联容器中都有的成员函数

- begin 返回指向容器中第一个元素的迭代器
- end 返回指向容器中最后一个元素后面的位置的迭代器
- rbegin 返回指向容器中最后一个元素的迭代器
- rend 返回指向容器中第一个元素前面位置的迭代器
- erase 从容器中删除一个或几个元素
- clear 从容器中删除所有元素

8.2.5 顺序容器的常用成员函数

- front 返回容器中第一个元素的引用
- back 返回容器中最后一个元素的引用
- push_back 在容器末尾增加新元素
- pop_back 删除容器末尾的元素
- erase 删除迭代器指向的元素（可能会使该迭代器失效），或删除一个区间，返回被删除元素后面的那个元素的迭代器

8.2.6 vector

vector 是可变长的动态数组。使用 vector 必须包含头文件 `#include <vector>`

vector 支持随机访问迭代器，可以根据下标随机访问某个元素的时间为常数，在尾部添加元素速度很快，在中间插入元素较慢。

所有的 STL 算法都可以对 vector 操作。

构造函数初始化：

- vector(); 无参构造函数，将容器初始化成空的
- vector(int n); 将容器初始化成有 n 个元素

- `vector(int n, const T &val);` 假定元素类型是 `T`，将容器初始化成有 `n` 个元素，每个元素的值都是 `val`
- `vector(iterator first, iterator last);` 将容器初始化为与别的容器上区间 `[first, last)` 一致的内容

其他常用函数：

- `void pop_back();` 删除容器末尾的元素
- `void push_back(const T &val);` 将 `val` 添加到容器末尾
- `int size();` 返回容器中第一个元素的引用
- `T &front();` 返回容器中第一个元素的引用
- `T &back();` 返回容器中最后一个元素的引用

8.2.7 list

`list` 上是数据结构中的双向链表。使用 `list` 需要包含头文件 `#include <list>`
在 `list` 中的任何位置插入或删除元素都是常数时间。`list` 不支持根据下标随机存取元素。
`list` 是顺序容器，具有所有顺序容器都有的成员函数。`list` 还支持 8 个成员函数：

- `push_front` 在链表最前面插入
- `pop_front` 删除链表最前面的元素
- `sort` 排序（`list` 不支持 STL 的算法 `sort`）
- `remove` 删除和指定值相等的所有元素
- `unique` 删除容器中的所有连续重复元素，仅仅留下每组等值元素中的第一个元素。
- `merge` 合并两个链表，并清空被合并的链表
- `reverse` 颠倒（逆序）链表
- `splice` 在指定位置前面插入另一链表中的一个或多个元素，并在另一链表中删除被插入的元素

`list` 容器的迭代器不支持完全随机访问，不能用 STL 中的 `sort` 函数进行排序。但 `list` 有自己的 `sort` 成员函数：

```
list<T> className;
className.sort(compare);    // compare can be defined by programmers
className.sort();           // no arguments, sort from smallest to largest
```

`list` 容器只能使用双向迭代器，不支持大于/小于比较运算符，`[]` 运算符和随机移动（即类似“list 的迭代器 + 2”这种操作）。

8.2.8 deque

deque 是双向队列，使用它时必须要包含头文件 `#include <deque>`

所有适用于 vector 的操作都适用于 deque。deque 还有 push_front（将元素插入到容器的头部）和 pop_front（删除头部的元素）操作。deque 没有 sort 成员函数。

8.2.9 set 和 multiset

set、multiset，map、multimap 都属于关联容器。关联容器内部元素有序排列，新元素插入的位置取决于它的值，查找速度快。

关联容器除了各容器都有的函数外，还支持以下成员函数：

- find 查找等于某个值的元素（x 小于 y 和 y 小于 x 同时不成立即为相等）
- lower_bound 查找某个下界
- uppper_bound 查找某个上界
- equal_range 同时查找上界和下界
- count 计算等于某个值的元素个数（x 小于 y 和 y 小于 x 同时不成立即为相等）
- insert 用于插入一个元素或一个区间

预备知识：pair 模板：

```
template <class _T1, class _T2>
struct pair {
    typedef _T1 first_type;
    typedef _T2 second _type;
    _T1 first;
    _T2 second;
    pair() : first(), second() {}
    pair(const _T1 &__a, const _T2 &__b) : first(__a), second(__b) {}
    template <class _U1, class _U2>
    pair(const pair<_U1, class _U2> &__p)
        : first(__p.first), second(__p.second) {}
};
```

map 和 multimap 容器里放着的都是 pair 模板类的对象，且按 first 从小到大排序。

上面代码中的第三个构造函数用法示例：

```
pair<int, int> p(pair<double, double>(5.5, 4.6));   // p.first = 5, p.second = 4
```

multiset 的定义：

```
template <class Key, class Pred = less<Key>, class A = allocator<Key> >
class multiset {...};
```

Pred 类型的变量决定了 multiset 中的元素，以及“一个比另一个小”的定义。multiset 运行过程中，比较连个元素 x 和 y 大小的做法，就是生成一个 Pred 类型的变量。假定该变量为 op，若表达式 op(x, y) 返回值为 true，则 x 比 y 小。

Pred 的缺省类型是 less<Key>，less 模板的定义如下：

```
template <class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T &x, const T &y) const {
        return x < y;   // less 模板是靠 < 来比较大小的
    }
};
```

multiset 的成员函数：

- iterator find(const T &val);
  在容器中查找值为 val 的元素，返回其迭代器。如果找不到，返回 end()。
- iterator insert(const T &val);
  将 val 插入到容器中并返回其迭代器。
- void insert(iterator first, iterator last);
  将区间 [first, last) 插入容器。
- int count(const T &val);
  统计有多少个元素的值和 val 相等。
- iterator lower_bound(const T &val);
  查找一个最大的位置 it，使得 [begin(), it) 中所有元素都比 val 小。
- iterator upper_bound(const T &val);
  查找一个最小的位置 it，使得 [it, end()) 中所有的元素都比 val 大。
- pair<iterator, iterator> equal_range(const T &val);
  同时求得 lower_bound 和 upper_bound。
- iterator erase(iterator it);
  删除 it 指向的元素，返回其后面的元素的迭代器（不同编译器可能结果不同）

multiset 的用法：

```
#include <set>
using namespace std;
class A {};
int main() {
    multiset<A> a;
    // error: invalid operands to binary expression ('const A' and 'const A')
    a.insert(A());
}
```

multiset<A> a; 等价于 multiset<A, less<A> > a; 插入元素时，multiset 会将被插入元素和已有元素进行比较。由于 less 模板是用 < 进行比较的，所以这都要求 A 的对象能用 < 比较，即适当重载了 < 运算符。

set 的定义：

```
template <class Key, class Pred = less<Key>, class A = allocator<Key> >
class set {...};
```

对于 set 来说，如果有两个元素 a 和 b，a < b 和 b < a 也不成立，那么它就会认为 a 和 b 是重复的。在 set 中插入已有元素时，会被 set 忽略。

8.2.10 map 和 multimap

前面已经提到了 pair 模板，map/multimap 里放着的都是 pair 模板类的对象，且按 first 从小到大排序。

multimap 的定义：

```
template <class Key, class T, class Pred = less<Key>, class A = allocator<T> >
class multimap {
    ...
    typedef pair<const Key, T> value_type;
    ...
};  // Key 代表关键字的类型
```

multimap 中的元素由 <key, value> 组成，每个元素是一个 pair 对象，关键字就是 first 成员变量，其类型是 Key。

multimap 中允许许多个元素的关键字相同。元素按照 first 成员变量从小到大排列，缺省情况下用 less<Key> 定义关键字的“小于”关系。

下面三段程序，哪个不会导致编译出错？（答案是第一个）

- ```
  multimap<string, greater<string> > mp;
  ```
- ```
  multimap<string, double, less<int> > mp1;
  mp1.insert(make_pair("ok", 3.14));
  ```
- ```
  multimap<string, double, less<int> > mp2;
  mp2.insert("ok", 3.14);
  ```
- 都会导致编译出错

map 的定义：

```
template <class Key, class T, class Pred = less<Key>, class A = allocator<T> >
class map {
    ...
    typedef pair<const Key, T> value_type;
};
```
```

map 中的元素都是 pair 模板类的对象。关键字（first 成员变量）各不相同。元素按照关键字从小到大排列，缺省情况下用 less<Key>，即“<”定义“小于”。

map 的 [] 成员函数：
若 pairs 为 map 模板类的对象，pairs[key] 返回对关键字等于 key 的元素的值（second 成员变量）的引用。若没有关键字为 key 的元素，则会往 pairs 里插入一个关键字为 key 的元素，其值用无参构造函数初始化，并返回其值的引用。

如 map<int, double> pairs; 则 pairs[50] = 5; 会修改 pairs 中关键字为 50 的元素，使其值变为 5。

由很多例子可以看出，关联容器特别适合用于一些需要不断更新数据和查询数据的情形。

8.2.11 容器适配器

容器适配器可以用某种顺序容器来实现（让已有的顺序容器以栈/队列的方式工作）。

1. stack 头文件 <stack>
   栈——后进后出
2. queue 头文件 <queue>
   队列——先进先出
3. priority_queue 头文件 <queue>
   优先级队列——最高优先级元素总是第一个出列

容器适配器都有 3 个成员函数：

- push 添加一个元素
- top 返回栈顶部或队头元素的引用
- pop 删除一个元素

容器适配器上没有迭代器。STL 中各种排序、查找、变序等算法都不适合容器适配器。

stack 是后进先出的数据结构，只能插入、删除、访问栈顶的元素。stack 可用 vector、list 或 deque 实现。在 STL 中，默认使用 deque 来实现 stack，用 vector 和 deque 实现要比用 list 实现性能好。

```
template <class T, class Cont = deque<T> >
class stack {
    ...
};
```

stack 中主要的三个成员函数：

- void push(const T &x);
  将 x 压入栈顶
- void pop();
  弹出（即删除）栈顶元素

- T &top();
  返回栈顶元素的引用。通过该函数可以读取栈顶元素的值，也可以修改栈顶元素。

queue 和 stack 基本类似，可以用 list 和 deque 实现。缺省情况下用 deque 实现：

```
template <class T, class Cont = deque<T> >
class queue {
    ...
};
```

queue 同样也有 push、pop 和 top 函数。push 发生在队尾；pop、top 发生在队头，先进先出。

priority_queue 和 queue 类似，可以用 vector 和 deque 实现，缺省情况下用的是 vector。

priority_queue 通常会采用堆排序的技术，以保证最大的元素总是在最前面。在执行 pop 操作，删除的是最大的元素；执行 top 操作时，返回的是最大元素的引用。

priority_queue 默认的元素比较器是 less<T>。

8.3 迭代器简介

迭代器用于指向顺序容器和关联容器中的元素。迭代器模拟了指针，可以用 * 或者 -> 运算符来访问容器中的元素。

迭代器有 const 和非 const 两种，通过迭代器可以读取它指向的元素。通过非 const 迭代器还能修改其指向的元素。

定义一个容器类（容器模板实例化后的类）的迭代器：容器类名::iterator 变量名;

或定义一个容器类的常量迭代器：容器类名::const_iterator 变量名; 可以通过此迭代器访问其指向的元素，但不可以修改它指向的元素。

或定义一个容器类的反向迭代器：容器类名::reverse_iterator 变量名; 可以通过此迭代器反向地访问其指向的元素。反向迭代器和迭代器的类型是不兼容的。

访问一个迭代器指向的元素：*迭代器变量名

迭代器上可以执行 ++ 操作，以使其指向容器中的下一个元素。如果迭代器到达了容器中的最后一个元素的后面，此时再使用它，就会出错，类似于使用 NULL 或未初始化的指针一样。

在 STL 中我们常用的迭代器有两种，一种是双向迭代器，另一种是随机访问迭代器。

- 支持双向迭代器的容器：list、set/multiset、map/multimap
- 支持随机访问迭代器的容器：vector、deque
- 不支持迭代器的容器：stack、queue、priority_queue

8.3.1 双向迭代器

若 p 和 p1 都是双向迭代器，则可对 p、p1 可进行以下操作：

- ++p / p++ 使 p 指向容器中下一个元素
- --p / p-- 使 p 指向容器中上一个元素
- *p 取 p 指向的元素

- `p = p1;` 赋值
- `p == p1 / p != p1` 判断是否相等、不等

8.3.2 随机访问迭代器

若 `p` 和 `p1` 都是随机访问迭代器，则可对 `p`、`p1` 进行以下操作：

- 双向迭代器的所有操作
- `p += i;` 将 `p` 向后移动 `i` 个元素
- `p -= i;` 将 `p` 向前移动 `i` 个元素
- `p + i` 值为指向 `p` 后面的第 `i` 个元素的迭代器
- `p - i` 值为指向 `p` 前面的第 `i` 个元素的迭代器
- `p[i]` 值为 `p` 后面第 `i` 个元素的引用
- `p < p1, p <= p1, p > p1, p >= p1`

有的算法，例如 `sort`、`binary_search` 需要通过随机访问迭代器来访问容器中的元素，那么 `list` 以及关联容器就不支持该算法。

8.3.3 利用迭代器遍历容器

`vector` 的迭代器是随机迭代器。遍历 `vector` 可以有以下几种做法（`deque` 亦然）：

```
vector<int> v(100);
int i;
for (i = 0; i < v.size(); i++) {
    v[i] = i;
    cout << v[i] << " ";   // randomly access by subscript
}
cout << endl;

vector<int>::const_iterator ii;
for(ii = v.begin(); ii != v.end(); ii++) {
    cout << *ii << " ";
}
cout << endl;

for (ii = v.begin(); ii < v.end(); ii++) {
    cout << *ii << " ";
}
cout << endl;
```

```
ii = v.begin();
while (ii < v.end()) {
    cout << *ii << " ";
    ii = ii + 2;
}
cout << endl;
```

list 的迭代器是双向迭代器，正确的遍历 list 的方法：

```
list<int> v;
list<int>::const_iterator ii;
for (ii = v.begin(); ii != v.end(); ++ii) {
    cout << *ii;
}
```

错误的遍历 list 的方法：

```
list<int> v;
list<int>::const_iterator ii;
// wrong, bidirectional iterators don't support <.
for (ii = v.begin(); ii < v.end(); ++ii) {
    cout << *ii;
}
// wrong, list doesn't have [] member function
for (int i = 0; i < v.size(); i++) {
    cout << v[i];
}
```

双向迭代器不支持 <，list 没有 [] 成员函数。

8.4 算法简介

算法就是一个个的函数模板，大多数在 <algorithm> 中定义。STL 中提供了能在各种容器中通用的算法，比如查找、排序等。

算法通过迭代器来操纵容器中的元素。许多算法，例如排序和查找算法，可以对容器中的一个局部区间进行操作，因此需要两个参数，一个是起始元素的迭代器，一个是终止元素后面一个元素的迭代器。

有的算法返回一个迭代器。比如 find() 算法，在容器中查找一个元素，并返回一个指向该元素的迭代器。

算法既可以处理容器，也可以处理普通数组。

find() 算法示例：

```
template<class InIt, class T>
InIt find(InIt first, InIt last, const T &val);
```

其中，first 和 last 这两个参数都是容器的迭代器，它们给出了容器中的查找区间起点和终点 [first, last)。区间的起点是位于查找范围之中的，而终点不是。

`find` 在 `[first, last)` 查找等于 `val` 的元素。`find()` 利用 `==` 运算符来判断是否相等，其返回值是一个迭代器。如果找到，则该迭代器指向被找到的元素；如果找不到，则该迭代器等于 `last`。

`find()` 的时间复杂度是 $O(n)$。

STL 中的算法大致可以分为以下七类：

- 不变序列算法
- 变值算法
- 删除算法
- 变序算法
- 排序算法
- 有序区间算法
- 数值算法

大多重载的算法都是有两个版本的：

- 用 `==` 判断元素是否相等，或用 `<` 来比较大小
- 多出一个类型参数 `Pred` 和函数形参 `Pred op`，通过表达式 `op(x, y)` 的返回值（`true/false`）来判断 `x` 是否“等于”`y`，或者 `x` 是否“小于”`y`。

如 `min_element` 就有如下两个版本：

- `iterator min_element(iterator first, iterator last);`
- `iterator min_element(ieterator first, iterator last, Pred op);`

其中，第一个版本利用 `<` 来比较大小；第二个版本利用 `Pred op` 来比较大小。在调用第二个版本的 `min_element` 时，传递给 `op` 的实参可以是函数名，也可以是函数对象。

8.4.1 不变序列算法

不变序列算法不会修改算法所作用的容器或对象，适用于顺序容器和关联容器，时间复杂度都是 $O(n)$。

常见的不变序列算法：

- `min`
  求两个对象中较小的（可自定义比较器）
- `max`
  求两个对象中较大的（可自定义比较器）
- `min_element`
  求区间中的最小值（可自定义比较器）

```
template <class FwdIt>
FwdIt min_element(FwdIt first, FwdIt last);
```

返回 [first, last) 中最小元素的迭代器，以 < 作比较器。最小指没有元素比它小，而不是它比别的不同元素都小，因为即便 a != b，a < b 和 b < a 有可能都不成立。

- max_element
  求区间的最大值（可自定义比较器）

```
template <class FwdIt>
FwdIt max_element(FwdIt first, FwdIt last);
```

  返回 [first, last) 中最大元素（不小于任何其他元素）的迭代器，以 < 作比较器

- for_each
  这里是不变序列的版本，对区间中的每个元素都做某种操作

```
template <class InIt, class Fun>
Fun for_each(InIt first, InIt last, Fun f);
```

  对 [first, last) 中的每个元素 e 执行 f(e)，要求 f(e) 不能改变 e

- count
  计算区间中等于某值的元素个数（利用 x == y 比较）

```
template <class InIt, class T>
size_t count(InIt first, InIt last, const T &val);
```

- count_if
  计算区间中符合某种条件的元素个数

```
template <class InIt, class Pred>
size_t count_if(InIt first, InIt last, Pred pr);
```

  计算 [first, last) 中复合 pre(e) == true 的元素 e 的个数

- find
  在区间中查找等于某值的元素

```
template <class InIt, class T>
InIt ifnd(InIt first, InIt last, const T &val);
```

  返回区间 [first, last) 中的迭代器 i，使得 *i == val

- find_if
  在区间中查找符合条件的元素

```
template <class InIt, class Pred>
InIt find_if(InIt first, InIt last, Pred pr);
```

  返回区间 [first, last) 中的迭代器 i，使得 pr(*i) == true

- find_end
  在区间中查找另一个区间最后一次出现的位置（可自定义比较器）

- `find_first_of`
  在区间中查找第一个出现在另一个区间中的元素（可自定义比较器）
- `adjacent_find`
  在区间中寻找第一次出现连续两个相等元素的位置（可自定义比较器）
- `search`
  在区间中查找另一个区间第一次出现的位置（可自定义比较器）
- `search_n`
  在区间中查找第一次出现等于某值的连续 `n` 个元素（可自定义比较器）
- `equal`
  判断两区间是否相等（可自定义比较器）
- `mismatch`
  逐个比较两个区间的元素，返回第一次发生不相等的两个元素的位置（可自定义比较器）
- `lexicographical_compare`
  按字典序比较两个区间的大小（可自定义比较器）

8.4.2 变值算法

变值算法会修改源区间或目标区间元素的值。但是被修改的区间不能属于关联容器，因为这可能会破坏关联容器的有序性。

常见的变值算法：

- `for_each`
  对区间中的每个元素都做某种操作
- `copy`
  复制一个区间到别处
- `copy_backward`
  复制一个区间到别处，但目标区间是从后往前被修改的
- `transform`
  将一个区间的元素变形后拷贝到另一个区间

  ```
  template <class InIt, class OutIt, class Unop>
  OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
  ```

  对 `[first, last)` 中的每个迭代器 `I`
  - 执行 `uop(*I)` 并将结果依次放入从 `x` 开始的地方
  - 要求 `uop(*I)` 不得改变 `*I` 的值

  该模板返回值是个迭代器，即 `x + (last - first)`，`x` 可以和 `first` 相等

- `swap_ranges`
  交换两个区间内容
- `fill`
  用某个值填充空间
- `fill_n`
  用某个值替换区间中的 `n` 个元素
- `generate`
  用某个操作的结果填充区间
- `generate_n`
  用某个操作的结果替换区间中的 `n` 个元素
- `replace`
  将区间中的某个值替换为另一个值
- `replace_if`
  将区间中符合某种条件的值替换成另一个值
- `replace_copy`
  将一个区间拷贝到另一个区间，拷贝时某个值要换成新值拷过去
- `replace_copy_if`
  将一个区间拷贝到另一个区间，拷贝时符合某条件的值要换成新值拷过去

在“Week9/ValueChangingAlgorithms.cpp”的示例代码中，`ostream_iterator<int> output(cout, " ");` 定义了一个 `ostream_iterator<int>` 对象，可以通过 `cout` 输出以“ ”（空格）分隔的一个个整数。之后，`copy(v.begin(), v.end(), output);` 会导致 `v` 的内容在 `cout` 上输出。

`copy` 函数模板（算法）

```
template <class InIt, class OutIt>
OutIt copy(InIt first, InIt last, OutIt x);
```

该函数对每个在区间 `[0, last - first)` 中的 `N` 执行一次 `*(x + N) = *(first + N)`，返回 `x + N`。

对于 `copy(v.begin(), v.end(), output);` `first` 和 `last` 的类型是 `vector<int>::const_iterator`，`output` 的类型是 `ostream_iterator<int>`。

`copy` 的源代码：

```
template <class _II, class _OI>
inline _OI copy(_II _F, _II _L, _OI _X) {
    for ( ; _F != _L; ++_X, ++_F)
        *_X =  *_F;
    return(_X);
}
```

观察下面的代码，考虑应该如何编写 `My_ostream_iterator` 才能使其完成预期的功能。

```
int a[4] = {1, 2, 3, 4};
My_ostream_iterator<int> oit(cout, "*");
copy(a, a + 4, oit);    // print 1*2*3*4
ofstream oFile("test.txt", ios::out);
My_ostream_iterator<int> oitf(oFile, "*");
copy(a, a + 4, oitf);   // write 1*2*3*4 into test.txt
oFile.close();
```

上面程序中调用语句 copy(a, a + 4, oit) 实例化后得到 copy 如下：

```
My_ostream_iterator<int> copy(int *_F, int *_L, My_ostream_iterator<int> _X) {
    for ( ; _F != _L; ++_X, ++_F)
        *_X = *_F;
    return(_X);
}
```

所以 My_ostream_iterator 应该对 ++、* 和 = 运算符进行重载。重载后的类模板如下：

```
template <class T>
class My_ostream_iterator : public iterator<output_iterator_tag, T> {
private:
    string sep;     // delimiter
    ostream &os;    // to store cout
public:
    My_ostream_iterator(ostream &o, string s) : sep(s), os(o) {}
    void operator++() {}
    My_ostream_iterator &operator*() {
        return *this;
    }
    My_ostream_iterator &operator=(const T &val) {
        os << val << sep;
        return *this;
    }
};
```

由以上示例可以看出，一个 STL 的模板究竟能实现哪些功能，并不是由它的名字决定的，而是取决于我们的想象力。

8.4.3 删除算法

删除算法可以用于删除一个容器里的某些元素。所谓“删除”，并不会使容器里的元素减少，而是将所有应该被删除的元素看作是空位子，用留下的元素从后往前移，依次去填空位子。元素往前移后，它原来的位置也会变成空位子，也应该由后面的留下的元素来填上。最后，没有被填上的空位子，维持其原来的值不变，可以被打印出来。

删除算法不应作用于关联容器。

常见的删除算法：

- `remove`
  删除区间中等于某个值的元素
- `remove_if`
  删除区间中满足某种条件的元素
- `remove_copy`
  拷贝区间到另一个区间。等于某个值的元素不拷贝
- `remove_copy_if`
  拷贝区间到另一个区间。符合某种条件的元素不拷贝
- `unique`
  删除区间中连续相等的元素，只留下一个（可自定义比较器）
- `unique_copy`
  拷贝区间到另一个区间。连续相等的元素，只拷贝第一个到目标区间（可自定义比较器）

以上的算法复杂度都是 $O(n)$ 的。
`unique` 类模板定义有两个版本，分别采用不同的比较大小的方式。

- 用 `==` 比较是否相等

```
template <class FwdIt>
FwdIt unique(FwdIt first, FwdIt last);
```

- 用 `Pred pr` 比较是否相等

```
template <class FwdIt, class Pred>
FwdIt unique(FwdIt first, FwdIt last, Pred pr);
```

  - 用 `pr(x, y)` 为 `true` 说明 `x` 和 `y` 相等
  - 对 `[first, last)` 这个序列中连续相等的元素，只留下第一个
  - 返回值时迭代器，指向元素删除后的区间的最后一个元素的后面

8.4.4 变序算法

变序算法改变容器中元素的顺序，但是不改变元素的值。变序算法不适用于关联容器。变序算法的复杂度都是 $O(n)$ 的。

常见的变序算法：

- `reverse`
  颠倒区间 `[first, last)` 前后次序

```
template <class BidIt>
void reverse(BidIt first, BidIt last);
```

- `reverse_copy`
  把一个区间颠倒后的结果拷贝到另一个区间，源区间不变。

- `rotate`
  将区间进行循环左移

- `rotate_copy`
  将区间以首尾相接的形式进行旋转后的结果拷贝到另一个区间，源区间不变

- `next_permutation`
  将区间改为下一个排列（排列是有大小的，这个大小可通过自定义比较器来设定）。如果能产生一个字典序更大的排列，就会返回 `true`；否则返回 `false`。

```
template <class InIt>
bool next_permutation(InIt first, InIt last);
```

- `prev_permutation`
  将区间改为上一个排列（可自定义比较器）

- `random_shuffle`
  随机打乱区间内元素的顺序

```
template <class RanIt>
void random_shuffle(RanIt first, RanIt last);
```

- `partition`
  把区间内满足某个条件的元素移到前面，不满足该条件的移到后面

- `stable_patition`
  把区间内满足某个条件的元素移到前面，不满足该条件的移到后面。而对这两部分的元素，分别保持它们原来的先后次序不变。

8.4.5 排序算法

排序算法比前面的变序算法的复杂度更高，一般是 $O(nlog(n))$。排序算法需要随机访问迭代器的支持，不适用于关联容器和 `list`。

常见的排序算法：

- `sort`
  将区间从小到大排序（可自定义比较器）

- `stable_sort`
  将区间从小到大排序，并保持相等元素间的相对次序（可自定义比较器）

- `partial_sort`
  对区间部分排序，直到最小的 `n` 个元素就位（可自定义比较器）

- `partial_sort_copy`
  将区间前 `n` 个元素的排序结果拷贝到别处，源区间不变（可自定义比较器）

- nth_element
  对区间部分排序，使得第 n 小的元素（n 从 0 开始算）就位，而且比它小的都在它前面，比它大的都在它后面。

- make_heap
  使区间成为一个“堆”（可自定义比较器）

- push_heap
  将元素加入一个“堆”区间（可自定义比较器）

- pop_heap
  从“堆”区间删除栈顶元素（可自定义比较器）

- sort_heap
  对一个“堆”区间进行排序，排序结束后，该区间就是普通的有序区间，不再是“堆”了（可自定义比较器）

sort 模板有两个版本，分别对应不同的比较大小的方式。
根据 < 进行比较的版本，默认按升序排序，判断 x 是否应该比 y 靠前，要看 x < y 是否为 true：

```
template <class RanIt>
void sort(RanIt first, RanIt last);
```

根据 Pred pr 进行比较的版本，按升序排序，判断 x 是否应该比 y 靠前，就看 pr(x, y) 的返回值是否为 true：

```
template <class RanIt, class Pred>
void sort(RanIt first, RanIt last, Pred pr);
```

sort 实际上是快速排序，时间复杂度为 $O(n*log(n))$。其平均性能最优，但是在“最坏情况”下，性能可能非常差，会达到 $O(n^2)$。

如果要保证“最坏情况”下的性能，那么可以使用 stable_sort，其用法与 sort 相同。stable_sort 实际上是归并排序，特点是能保持相等元素之间的先后次序。在有足够存储空间的情况下，复杂度为 $O(n*log(n))$，否则复杂度为 $O(n*log(n)*log(n))$。

排序算法要求随机存取迭代器的支持，所以 list 不能使用排序算法，要使用 list::sort。

8.4.6 有序区间算法

有序区间算法要求所操作的区间是已经从小到大排好序的。有序区间算法需要随机访问迭代器的支持，不能用于关联容器和 list。

常见的有序区间算法：

- binary_search
  判断区间中是否包含某个元素。采用折半查找的方法，时间复杂度为 $O(log(n))$。要求容器已经有序且支持随机访问迭代器，返回是否找到。

```
// 用 < 作比较器
template <class FwdIt, class T>
```

```
bool binary_search(FwdIt first, FwdIt last, const T &val);
// 用 Pred pr 作比较器，若 p(x, y) 为 true，则认为 x 小于 y
template <class FwdIt, class T, class Pred>
bool binary_search(FwdIt first, FwdIt last, const T &val, Pred pr);
```

- includes
  判断是否一个区间中的每个元素，都在另一个区间中

```
// 用 < 作比较器
template <class InIt1, class InIt2>
bool includes(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2);
// 用 Pred pr 作比较器，pr(x, y) == true 说明 x 和 y 相等
template <class InIt1, class InIt2>
bool includes(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, Pred pr);
```

- lower_bound
  查找最后一个不小于某值的元素的位置

```
template <class FwdIt, class T>
FwdIt lower_bound(FwdIt first, FwdIt last, const T &val);
```

  要求 [first, last) 是有序的，查找 [first, last) 中最大的位置 FwdIt，使得 [first, FwdIt) 中所有的元素都比 val 小

- upper_bound
  查找第一个大于某值的元素的位置

```
template <class FwdIt, class T>
FwdIt upper_bound(FwdIt first, FwdIt last, const T &val);
```

  要求 [first, last) 是有序的，查找 [first, last) 中最小的位置 FwdIt，使得 [FwdIt, last) 中所有的元素都比 val 大

- equal_range
  同时获取 lower_bound 和 upper_bound

```
template <class FwdIt, class T>
pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T &val);
```

  要求 [first, last) 是有序的，返回值是一个 pair，假设为 p，则：
  - [first, p.first) 中的元素都比 val 小
  - [p.second, last) 中的所有元素都比 val 大
  - p.first 就是 lower_bound 的结果
  - p.last 就是 upper_bound 的结果

- merge
  合并两个有序区间到第三个区间：

```
// 用 < 作比较器
template <class InIt1, class InIt2, class OutIt>
OutIt merge(InIt1 first1, InIt last1, InIt first2, InIt2 last2, OutIt x);
// 用 Pred pr 作比较器
template <class InIt1, class InIt2, class OutIt, class Pred>
OutIt merge(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
    Pred pr);
```

把 [first1, last1)、[first2, last2) 两个升序序列合并，形成第 3 个升序序列，第 3 个升序序列以 x 开头

- set_union
  将两个有序区间的并拷贝到第三个区间

```
// 用 < 作比较器
template <class InIt1, class InIt2, class OutIt>
OutIt set_union(InIt1 first1, InIt last1, InIt first2, InIt2 last2, OutIt x);
// 用 Pred pr 作比较器
template <class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x,
    Pred pr);
```

- set_intersection
  将两个有序区间的交拷贝到第三个区间

```
// 利用 < 作比较器
template <class InIt1, class InIt2, class OutIt>
OutIt set_intersection(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2,
    OutIt x);
// 利用 Pred pr 作比较器
template <class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_intersection(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2,
    OutIt x, Pred pr);
```

求出 [first1, last1) 和 [first2, last2) 中共有的元素，放到从 x 开始的地方。若某个元素 e 在 [first1, last1) 里出现 n1 次，在 [first2, last2) 里出现 n2 次，则该元素在目标区间里出现 min(n1, n2) 次

- set_difference
  将两个有序区间的差拷贝到第三个区间

```
// 用 < 作比较器
template <class InIt1, class InIt2, class OutIt>
OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2,
    OutIt x);
```

```
// 用 Pred pr 作比较器
template <class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_difference(InIt1 first1, InIt1 last, InIt2 first2, InIt2 last2,
    OutIt x, Pred pr);
```

求出 [first1, last1) 中不在 [first2, last2) 中的元素，放到从 x 开始的地方。如果 [first1, last1) 里有多个相等元素不在 [first2, last2) 中，则这多个元素也都会被放入 x 代表的目标区间里

- set_symmetric_difference
  将两个有序空间的对称差（并减去交）拷贝到第三个区间

```
// 利用 < 作比较器
template <class InIt1, class InIt2, class OutIt>
OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2,
    InIt2 last2, OutIt x);
// 利用 Pred pr 作比较器
template <class InIt1, class InIt2, class OutIt, class Pred>
OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2,
    InIt2 last2, OutIt x, Pred pr);
```

把两个区间里相互不在另一区间里的元素放入 x 开始的地方

- inplace_merge
  将两个连续的有序区间原地合并为一个有序区间

8.5 STL 中“大”、“小”和“相等”的概念

关联容器内部的元素是从小到大排序的。“大”和“小”的含义是程序员自定义的。

有些算法要求其操作的区间是从小到大排序的，称为“有序区间算法”，例如 binary_search。有些算法会对区间进行从小到大排序，称为“排序算法”，例如 sort。还有一些其他算法会用到“大”、“小”的概念。

使用 STL 时，在缺省的情况下，以下三个说法等价：

1. x 比 y 小
2. 表达式 x < y 为真
3. y 比 x 大

有时，“x 和 y 相等”等价于“x == y 为真”。例如在未排序的区间上进行的顺序查找算法 find。

有时，“x 和 y 相等”等价于“x 小于 y 和 y 小于 x 同时为假”。例如有序区间算法中，binary_search 关联容器自身的成员函数 find。

8.6 bitset

```
template<size_t N>
class bitset {
```

```
    ...
};
```

实际使用的时候，N 是个整型常数。例如：

```
bitset<40> bst;
```

其中 bst 是一个由 40 位组成的对象，用 bitset 的函数可以方便地访问任何一位。

bitset 的成员函数：

- `bitset<N> &operator&=(const bitset<N> &rhs);`
- `bitset<N> &operator|=(const bitset<N> &rhs);`
- `bitset<N> &operator^=(const bitset<N> &rhs);`
- `bitset<N> &operator<<=(size_t, num);`
- `bitset<N> &operator>>=(size_t, num);`
- `bitset<N> &set();` 全部置 1
- `bitset<N> &set(size_t pos, bool val = true);` 将某位置 1
- `bitset<N> &reset();` 全部置 0
- `bitset<N> &reset(size_t pos);` 将某位置 0
- `bitset<N> &flip();` 全部翻转
- `bitset<N> &flip(size_t pos);` 翻转某位
- `reference operator[](size_t, pos);` 返回对某位的引用
- `bool operator[](size_t pos) const;` 判断某位是否为 1
- `reference at(size_t, pos);`
- `bool at(size_t pos) const;`
- `unsigned long to_ulong() const;` 转换为整数
- `string to_string() const;` 转换成字符串
- `size_t count() const;` 计算 1 的个数
- `size_t size() const;`
- `bool operator==(const bitset<N> &rhs) const;`
- `bool operator!=(const bitset<N> &rhs) const;`
- `bool test(size_t pos) const;` 测试某位是否为 1
- `bool any() const;` 是否有某位为 1

- `bool none() const;` 是否全部为 0
- `bitset<N> operator<<(size_t pos) const;`
- `bitset<N> operator>>(size_t pos) const;`
- `bitset<N> operator~();`
- `static const size_t bitset_size = N;`

8.7 函数对象

若一个类重载了运算符 ()，则该类的对象就成为函数对象。

```
class CMyAverage {
public:
    double operator()(int a1, int a2, int a3) {
        return (double)(a1 + a2 + a3) / 3;
    }
};

int main(int argc, char const *argv[]) {
    CMyAverage average;
    cout << average(3, 2, 3) << endl;   // average.operator()(3, 2, 3)
    return 0;
}
```

8.7.1 函数对象的应用

Dev C++ 中的 accumulate 源代码 1:

```
template <typename _InputIterator, typename _Tp>
_Tp accumulate(_InputIterator __first, _InputIterator __last, _Tp __init) {
    for ( ; __first != __last; ++__first) {
        __init = __init + *__first;
        return __init;
    }
}
```

Dev C++ 中的 accumulate 源代码 2:

```
template <typename _InputIterator, typename _Tp, typename _BinaryOperation>
_Tp accumulate(_InputIterator __first, _InputIterator __last,
    _Tp __init, _BinaryOperation __binary_op) {
    for ( ; __first != __last; ++__first) {
        __init = __binary_op(__init, *__first);
        return __init;
```

```
        }
    }
```

在调用 accumulate 时，和 __binary_op 对应的实参可以是个函数或函数对象。

例如，现定义了如下函数，分别用于求一个数和另一个数的 n 次幂的和。并在 main 函数中利用 accumulate 调用它们，实现求一组数的 n 次幂之和。

```
int sumSquares(int total, int value) {  // compute total + value ^ 2
    return total + value * value;
}

template <class T>
class SumPowers {
private:
    int power;
public:
    SumPowers(int p) : power(p) {}
    // compute total + value ^ power
    const T operator()(const T &total, const T &value) {
        T v  = value;
        // compute (v ^ power)
        for (int i = 0; i < power -  1; ++i) {
            v = v * value;
        }
        return total + v;
    }
};

int main(int argc, char const *argv[]) {
    const int SIZE = 10;
    int a1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> v(a1, a1 + SIZE);
    cout << "1) ";
    printInterval(v.begin(), v.end());
    // 1) 1 2 3 4 5 6 7 8 9 10
    int result = accumulate(v.begin(), v.end(), 0, sumSquares);
    cout << "2) Sum of square: " << result << endl;
    // 2) Sum of square: 385
    result = accumulate(v.begin(), v.end(), 0, SumPowers<int>(3));
    cout << "3) Sum of cube: " << result << endl;
    // 3) Sum of cube: 3025
    result = accumulate(v.begin(), v.end(), 0, SumPowers<int>(4));
    cout << "4) Sum of quartic: " << result << endl;
```

```
    // 4) Sum of quartic: 25333
    return 0;
}
```

int result = accumulate(v.begin(), v.end(), 0, sumSquares); 会实例化出：

```
int accumulate(vector<int>::iterator first, vector<int>::iterator last,
    int init, int (*op)(int, int)) {    // op is a function pointer
    for ( ; first != last; ++first)
        init = op(init, *first);
    return init;
}
```

result = accumulate(v.begin(), v.end(), 0, SumPowers<int>(3)); 会实例化出：

```
int accumulate(vector<int>::iterator first, vector<int>::iterator last,
    int init, SumPowers<int> op) {      // op is a function object
    for ( ; first != last; ++first)
        init = op(init, *first);
    return init;
}
```

以上代码详见 “Week8/FunctionObjectSample.cpp” 文件。
以下模板可以用来生成函数对象，在使用时需要包含头文件 <functional>：

- equal_to
- greater
- less
- ...

8.7.2 greater 函数对象类模板及其应用

```
template <class T>
struct greater : public some_binary_function<T, T, bool> {
    bool operator()(const T &x, const T &y) const {
        return x > y;
    }
}
```

list 有两个 sort 成员函数：

- void sort();
  将 list 中的元素按“<”规定的比较方法升序排列。
- ```
  template <class Compare>
  void sort(Compare op);
  ```
  ```

将 list 中的元素按 op 规定的比较方法升序排列。即要比较 x、y 大小时，看 op(x, y) 的返回值，为 true 则认为 x 小于 y。

下面的代码利用 greater 实现了对 list 进行降序排列：

```
class MyLess {
public:
    bool operator()(const int &c1, const int &c2) {
        return (c1 % 10) < (c2 % 10);
    }
};

template <class T>
void print(T first, T last) {
    for ( ; first != last; ++first) {
        cout << *first << ", ";
    }
}

int main(int argc, char const *argv[]) {
    const int SIZE_B = 5;
    int a[SIZE_B] = {5, 21, 14, 2, 3};
    list<int> lst(a, a + SIZE_B);
    lst.sort(MyLess()); // MyLess() constructs a temporary object without arguments
    print(lst.begin(), lst.end());
    // 21, 2, 3, 14, 5,
    cout << endl;
    lst.sort(greater<int>());   // greater<int> constructs a temporary object
    print(lst.begin(), lst.end());
    // 21, 14, 5, 3, 2,
    cout << endl;
    return 0;
}
```

由此可见，关联容器和 STL 中许多算法，都是可以用函数或函数对象自定义比较器的。在自定义了比较器 op 的情况下，以下三种说法是等价的：

- x 小于 y
- op(x, y) 返回值为 true
- y 大于 x

9 强制类型转换

9.1 static_cast

static_cast 用来进行比较“自然”和低风险的转换，比如整型和实数型、字符型之间的互相转换。

static_cast 不能在不同类型的指针之间互相转换，也不能用于整型和指针之间的互相转换，也不能用于同类型的引用之间的转换。

9.2 reinterpret_cast

reinterpret_cast 用来进行各种不同类型的指针之间的转换、不同类型的引用之间的转换、以及指针和能容纳得下指针的整数类型之间的转换。转换的时候，执行的是逐个比特拷贝的操作。

9.3 const_cast

用来进行去除 const 属性的转换。将 const 引用转换成同类型的非 const 引用，将 const 指针转换为同类型的非 const 指针时用它。例如：

```
const string s = "Inception";
string &p = const_cast<string &>(s);
string *ps = const_cast<string *>(&s);  // the type of &s is const string *
```

9.4 dynamic_cast

dynamic_cast 专门用于将多态基类（包含虚函数的基类）的指针或引用，强制转换为派生类的指针或引用，而且能够检查转换的安全性。对于不安全的指针转换，转换结果返回 NULL 指针。

dynamic_cast 不能用于将非多态基类的指针或引用，强制转换为派生类的指针或引用。

dynamic_cast 也可以用于基类的引用到派生类的引用的强制转换，并且可以判断转换是否安全：

```
Derived &r = dynamic_cast<Derived &>(b);
```

如何判断转换是否安全呢？答案是不安全时抛出异常。

10 异常处理

程序运行中总难免发生错误，而引起这些错误异常情况的原因又多种多样。我们总希望在发生异常情况时不只是简单地终止程序运行，而是能够反馈异常情况的信息，能够对程序中已发生的事情做些处理，例如取消对输入文件的改动、释放已经申请的系统资源等。

通常的做法是：在预计会发生异常的地方，加入相应的代码，但这种做法并不总是适用的。例如：

```
... // 对文件 A 进行了相关的操作
fun(arg, ...);  // 可能发生异常
...
```

那么调用者应该如何知道 fun(arg, ...) 是否发生异常呢？如果 fun(arg, ...) 是别人已经开发好的代码，fun(arg, ...) 的编写者不知道其他人会如何使用这个函数，通过返回值来返回异常不符合编程的习惯，异常有很多种时通过返回值判断也很麻烦。

所以，需要一种手段，可以把异常与函数的接口分开，并且能够区分不同的异常。并且能够在函数体外捕获所发生的异常，并提供更多的异常信息。

10.1 用 try、catch 处理异常

```
double m, n;
cin >> m >> n;
try {
    cout << "before dividing." << endl;
    if (n == 0) {
        throw -1;
    } else {
        cout << m / n << endl;
    }
    cout << "after dividing" << endl;
} catch (double d) {
    cout << "catch(double) " << d << endl;
} catch (int e) {
    cout << "catch(int) " << e << endl;
}
cout << "finished" << endl;
```

如果在上面的程序运行时输入 9 0，try 模块就会捕获到这个异常，并通过 throw 将异常抛出，交给 catch 处理。

10.2 异常的再抛出

如果一个函数在执行的过程中，抛出的异常在本函数内就被 catch 块捕获处理了，那么该异常就不会抛给这个函数的调用者（也称“上一层的函数”）；如果异常在本函数中没被处理，就会抛给上一层的函数。

```
class CException {
public:
    string msg;
    CException(string s) : msg(s) {}
};

double divide(double x, double y) {
    if (y == 0) {
        throw CException("divided by zero");    // the exception is not handled here
    }
    cout << "in divide" << endl;
    return x / y;
```

```
}

int countTax(int salary) {
    try {
        if (salary < 0) {
            throw -1;
        }
        cout << "counting tax" << endl;
    } catch (int) {
        cout << "salary < 0" << endl;
    }
    cout << "tax counted" << endl;
    return salary * 0.15;
}

int main(int argc, char const *argv[]) {
    double f = 1.2;
    try {
        countTax(-1);       // handle the exception in countTax() function
        // salary < 0
        // tax counted
        f = divide(3, 0);   // handle the exception out of divide() function
        cout << "end of try block" << endl; // not printed
    } catch (CException e) {
        cout << e.msg << endl;
        // divided by zero
    }
    cout << "f = " << f << endl;
    cout << "finished" << endl;
    return 0;
}
```

10.3 C++ 标准异常类

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来的。常用的几个异常类有：bad_typeid、bad_cast、bad_alloc、ios_base::failure、logic_error。其中 logic_error 又派生出了 out_of_range 异常。

10.3.1 bad_cast

在用 dynamic_cast 进行从多态基类对象（或引用），到派生类的引用的强制类型转换时，如果转换是不安全的，则会抛出此异常。

```
class Base {
    virtual void func() {}
};

class Derived : public Base {
public:
    void print() {}
};

void printObj(Base &b) {
    try {
        // if not safe, it will throw bad_cast
        Derived &rd = dynamic_cast<Derived &>(b);
        rd.print();
    } catch (bad_cast &e) {
        cerr << e.what() << endl;
    }
}

int main(int argc, char const *argv[]) {
    Base b;
    printObj(b);    // std::bad_cast
    return 0;
}
```

10.3.2 bad_alloc

在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常。

```
try {
    char *p = new char[0x7fffffffffffff];
    delete[] p;
} catch (bad_alloc &e) {
    cerr << e.what() << endl;   // std::bad_alloc
}
```

10.3.3 out_of_range

用 vector 或 string 的 at 成员函数根据下标访问元素时，如果下标越界，就会抛出此异常。

```
vector<int> v(10);
try {
    v.at(100) = 100;                // throw out_of_range exception
} catch (out_of_range &e) {
```

```
    cerr << e.what() << endl;       // vector
}
string s = "hello";
try {
    char c = s.at(100);
} catch (out_of_range &e) {
    cerr << e.what() << endl;       // basic_string
}
```

11 C++11 特性

C++11 是 C++ 标准委员会在 2011 年发布的 C++ 标准，C++11 引入了很多新特性。

11.1 统一的初始化方法

```
struct A {
    int i, j;
    A(int m, int n) : i(m), j(n) {}
};

A func(int m, int n) {
    return {m, n};      // not recommended
}

int main(int argc, char const *argv[]) {
    int arr[3]{1, 2, 3};
    vector<int> iv{1, 2, 3};
    map<int, string> mp{{1, "a"}, {2, "b"}};
    string str{"Hello world"};
    int *p = new int[20]{1, 2, 3};  // left elements are 0
    A *pa = new A{3, 7};
    delete pa;
    return 0;
}
```

11.2 成员变量默认初始值

```
class B {
public:
    int m = 1234;
    int n;
};
```

```
int main(int argc, char const *argv[]) {
    B b;
    cout << b.m << endl;    // 1234
    return 0;
}
```

11.3 auto 关键字

auto 关键字用于定义变量，编译器可以自动判断变量的类型。

```
class A {
    A operator+(int n, const A &a) {
        return a;
    }
};

template <class T1, class T2>
auto add(T1 x, T2 y) -> decltype(x + y) {   // return type depends on (x + y)
    return x + y;
}

int main(int argc, char const *argv[]) {
    auto i = 100;       // i is int
    auto p = new A();   // p is A *
    auto k = 34343LL;   // k is long long
    map<string, int, greater<string> > mp;
    for (auto i = mp.begin(); i != mp.end(); ++i) {
        // i is map<string, int, greater<string> >::iterator
        cout << i->first << ", " << i->second;
    }
    auto d = add(100, 1.5); // double d = 101.5
    auto f = add(100, A()); // the type of f is A
    return 0;
}
```

11.4 decltype 关键字

可以通过 decltype 返回表达式的类型。

```
int i;
double t;
struct A {
```

```
    double x;
};
const A *a = new A();

decltype(a) x1;             // x1 is A *
decltype(i) x2;             // x2 is int
decltype(a->x) x3;          // x3 is double
decltype((a->x)) x4 = t;    // x4 is double &
// run with command '| c++filt'
cout << typeid(x1).name() << endl;  // main::A const*
cout << typeid(x2).name() << endl;  // int
cout << typeid(x3).name() << endl;  // double
cout << typeid(x4).name() << endl;  // double
```

11.5 智能指针 shared_ptr

智能指针 shared_ptr 是一个类模板，在使用时需要包含头文件 <memory>。

通过 shared_ptr 的构造函数，可以让 shared_ptr 对象托管一个 new 运算符返回的指针，写法为：

```
shared_ptr<T> ptr(new T);   // T can be int, char or class name etc.
```

此后 ptr 就可以像 T * 类型的指针一样来使用，即 *ptr 就是用 new 动态分配的那个对象，而且不必操心释放内存的事情。

多个 shared_ptr 对象可以同时托管一个指针，系统会维护一个托管计数。当无 shared_ptr 托管该指针时，delete 该指针。

shared_ptr 对象不能托管指向动态分配内存的数组的指针，否则程序运行会出错。

```
struct A {
    int n;
    A(int v = 0) : n(v) {}
    ~A() {
        cout << n << " destructor" << endl;
    }
};

int main(int argc, char const *argv[]) {
    shared_ptr<A> sp1(new A(2));    // sp1 owns A(2)
    shared_ptr<A> sp2(sp1);         // sp2 owns A(2) too
    cout << "1) " << sp1->n << ", " << sp2->n << endl;  // 1) 2, 2
    shared_ptr<A> sp3;
    A *p = sp1.get();               // p gets the pointer sp1 owns, i.e. A(2)
    cout << "2) " << p->n << endl;  // 2) 2
    sp3 = sp1;                      // sp3 owns A(2) too
```

```
    cout << "3) " << (*sp3).n << endl;  // 3) 2
    sp1.reset();                       // sp1 doesn't own A(2) anymore
    if (!sp1) {
        cout << "4) sp1 is null" << endl;   // 4) sp1 is null
    }
    A *q = new A(3);
    sp1.reset(q);                      // sp1 owns q, i.e. A(3)
    cout << "5) " << sp1->n << endl;         // 5) 3
    shared_ptr<A> sp4(sp1);            // sp4 owns A(3)
    sp1.reset();                       // sp1 doesn't own A(3) anymore
    cout << "before end main" << endl;
    sp4.reset();                       // sp4 doesn't own A(3) anymore
    // 3 destructor
    cout << "end main" << endl;
    // 2 destructor
    return 0;
}
```

在使用 shared_ptr 时需要注意，通过 reset() 函数托管一个指针时，并不会增加其他 shared_ptr 对象关于该指针的托管计数，在该指针消亡时，该指针有可能被 delete 多次，从而导致程序崩溃。

11.6 空指针 nullptr

```
int *p1 = NULL;
int *p2 = nullptr;
shared_ptr<double> p3 = nullptr;
if (p1 == p2) {
    cout << "p1 equals p2" << endl;
    // p1 equals p2
}
if (p3 == nullptr) {
    cout << "p3 equals nullptr" << endl;
    // p3 equals nullptr
}
// error: invalid operands to binary expression ('int *' and 'shared_ptr<double>')
// if (p2 == p3) {
//     cout << "p2 equals p3" << endl;
// }
if (p3 == NULL) {
    cout << "p3 equals NULL" << endl;
    // p3 equals NULL
}
```

```
bool b = nullptr;   // b = false
// error: cannot initialize a variable of type 'int' with an rvalue of type 'nullptr_t'
// int i = nullptr;
```

11.7 基于范围的 for 循环

```
int array[] = {1, 2, 3, 4, 5};
for (int & e: array) {
    e *= 10;
}
for (int e : array) {
    cout << e << ", ";
}
// 10, 20, 30, 40, 50,
cout << endl;
vector<A> st(array, array + 5);
for (auto & it : st) {
    it.n *= 10;
}
for (A it : st) {
    cout << it.n << ", ";
}
// 100, 200, 300, 400, 500,
cout << endl;
return 0;
```

11.8 右值引用和 move 语义

右值：一般来说，不能取地址的表达式，就是右值，能取地址的，就是左值。之前遇到的引用都是左值引用。

```
class A{};
A &r = A();     // error, A() 是无名变量，是右值
A &&r = A();    // OK, r 是右值引用
```

右值引用的主要目的是提高程序运行的效率，减少需要进行深拷贝的对象进行深拷贝的次数。

move 是 STL 中定义好的函数模板，可以把一个左值变为一个右值。在操作某些临时变量时，可以通过 move 语义来减少深拷贝的次数。

```
class String {
public:
    char *str;
    String() : str(new char[1]) {
        str[0] = 0;
```

```
    }
    String(const char *s) {
        str = new char[strlen(s) + 1];
        strcpy(str, s);
    }
    String(const String &s) {
        cout << "copy constructor is called" << endl;
        str = new char[strlen(s.str) + 1];
        strcpy(str, s.str);
    }
    String &operator=(const String &s) {
        cout << "copy operator= is called" << endl;
        if (str != s.str) {
            delete[] str;
            str = new char[strlen(s.str) + 1];
            strcpy(str, s.str);
        }
        return *this;
    }
    // move constructor
    String(String &&s) : str(s.str) {
        cout << "move constructor is called" << endl;
        s.str = new char[1];
        s.str[0] = 0;
    }
    // move assignment
    String &operator=(String &&s) {
        cout << "move operator= is called" << endl;
        if (str != s.str) { // check if it is self-assignment
            str = s.str;
            s.str = new char[1];
            s.str[0] = 0;
        }
        return *this;
    }
    ~String() {
        delete[] str;
    }
};
```

11.9 无序容器（哈希表）

无序容器也称为哈希表，属于关联容器，可以存储多对键值和映射值，并且允许通过键值快速搜索映射值。哈希表插入和查询的时间复杂度几乎是常数。使用时需包含 <unordered_map> 头文件。

```
unordered_map<string, int> turingWinner;
turingWinner.insert(make_pair("Dijkstra", 1972));
turingWinner.insert(make_pair("Scott", 1976));
turingWinner.insert(make_pair("Wilkes", 1967));
turingWinner.insert(make_pair("Hanmming", 1968));
turingWinner["Ritchie"] = 1983;
string name;
cin >> name;
unordered_map<string, int>::iterator p = turingWinner.find(name);
if (p != turingWinner.end()) {
    cout << p->second << endl;
} else {
    cout << "Not found" << endl;
}
```

11.10 正则表达式

正则表达式使用单个字符串来描述、匹配一系列的匹配某个句法规则的字符串。在 C++ 中使用正则表达式需要包含 <regex> 头文件。

```
regex reg("b.?p.*k");
    // . matches any single character
    // ? indicates zero or one occurrences of the preceding element
    // * indicates zero or more occurrences of the preceding element
    cout << regex_match("bopggk", reg) << endl;             // 1 (matched)
    cout << regex_match("boopgggk", reg) << endl;           // 0 (not matched)
    cout << regex_match("b pk", reg) << endl;               // 1 (matched)
    regex reg2("\\d{3}([a-zA-Z]+).(\\d{2}|N/A\\s\\1)");
    // \\d matches any signle digit
    // {n} the preceding item is matched exactly n times
    // () defines a marked subexpression
    // [] matches a single character that is contained within the brackets
    // + indicates one or more occurrences of the preceding element
    // | matches either the expression before or the expression after the operator
    // \\s matches a whitespace character
    // \\n matches what the nth marked subexpression matched, n is a digit from 1 to 9
    string correct = "123Hello N/A Hello";
    string incorrect = "123Hello 12 hello";
```

```
cout << regex_match(correct, reg2) << endl;          // 1 (matched)
cout << regex_match(incorrect, reg2) << endl;        // 0 (not matched)
```

11.11 Lambda 表达式

只使用一次的函数对象，能否不要专门为其编写一个类？只调用一次的简单函数，能否在调用时才写出其函数体？Lambda 表达式可以解决上述问题。

Lambda 表达式是一个匿名函数，即没有函数名的函数，基于数学中的 λ 演算得名。

Lambda 表达式的形式：

```
[外部变量访问方式说明符](参数表) -> 返回值类型 {
    语句组
}
```

其中，“外部变量访问方式说明符”可以是：

- `[=]` 以传值的形式使用所有外部变量
- `[]` 不使用任何外部变量
- `[&]` 以引用形式使用所有外部变量
- `[x, &y]` x 以传值形式使用，y 以引用形式使用
- `=, &x, &y` x, y 以引用形式使用，其余变量以传值形式使用
- `&, x, y` x, y 以传值的形式使用，其余变量以引用形式使用

“-> 返回值类型”也可以没有，没有则编译器自动判断返回值类型。

```
int x = 100, y = 200, z = 300;
cout << [](double a, double b) { return a + b; } (1.2, 2.5) << endl;    // 3.7
auto ff = [=, &y, &z](int n) {
    cout << x << endl;              // 100
    y++;
    z++;
    return n * n;
};
cout << ff(15) << endl;             // 225
cout << y << ", " << z << endl;     // 201, 301

int a[4] = {4, 2, 11, 33};
sort(a, a + 4, [](int x, int y) -> bool { return x % 10 < y % 10; });
for_each(a, a + 4, [](int x) {cout << x << " ";});  // 11 2 33 4
cout << endl;

int n = 0;
```

```
int b[] = {1, 2, 3, 4};
for_each(b, b + 4, [&](int e) { ++e; n += e; });
cout << n << ", " << b[2] << endl;  // 14, 3

vector<int> c{1, 2, 3, 4};
int total = 0;
for_each(c.begin(), c.end(), [&](int &x) { total += x; x *= 2; });
cout << total << endl;              // 10
for_each(c.begin(), c.end(), [](int x) {cout << x << " ";});    // 2 4 6 8
cout << endl;
```

利用 Lambda 表达式实现递归求斐波那契数列的第 n 项：

```
function<int(int)> fib = [&fib](int n) {
    return n <= 2 ? 1 : fib(n - 1) + fib(n - 2);
};
cout << fib(5) << endl;             // 5
```

`function<int(int)>` 表示返回值为 `int`、有一个 `int` 参数的函数。